JCrasher: an automatic robustness tester for Java

Nuno Palma 86903
Paulo Bolinhas 110976
Rui Martins 110890

Introduction

Research Work

Formal specifications, program mutation, program analysis to derive constraints and white-box coverage-based testing

Pratical Settings

Black box and static regression testing. But these are all manually

What is the Alternative?

Random Testing

Random Testing

- Provides well-formed but random data as input
- Checks if the results are correct (with limited human interaction)
- Cheap (only requires human input when a potential error is found)
- Easily covers edge boundaries

JCrasher

- Generates random but type-correct inputs
- Tries to make the application crash
- Exercises pre-conditions on public methods (interface to the world)
- Can run unsupervised on a separate machine in parallel with regular development

JCrasher vs Related Work

Enhanced JUnit

Does not take into consideration the Java Type System, JCrasher does

Native Java

Defines heuristics for determining whether a Java exception should be considered a program bug or the JCrasher supplied inputs have violated the code's preconditions. This is particularly important for Java because of the lack of concrete method preconditions in the language.

Clean State Testing

Runs on a clean state: changes to static data by previous tests do not affect the current test.

Eclipse Integration

Produce test files for JUnit, which can be integrated into Eclipse IDE. This eases the use and inspection of tests and allows to permanently integrate the test suite if it is good enough

Overview and Assumptions

JCrasher Inputs

- Takes Java Byte Code as input
- The user can specify constraints
  - Maximum Depth

Class and Method Inputs

- Examines the parameters
- Computes the possible values
- Map from Type to
  - Pre-Set values or
  - Methods that return that Type

JCrasher Robustness

- Ex: *“A class should not crash with an unexpected runtime exception, regardless of the parameters provided.“*
- Determines if an exception constitutes an error or not

Public Methods

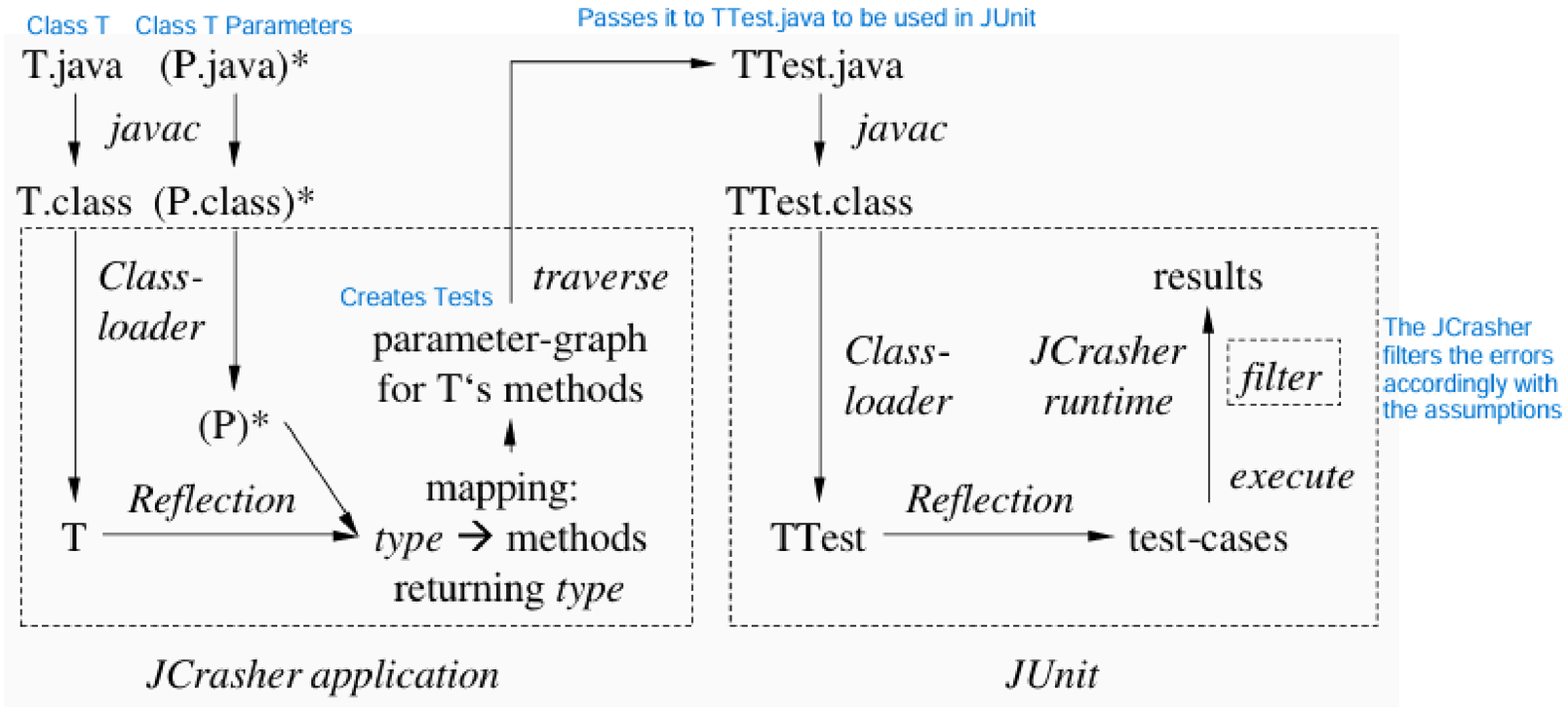
Inputs

Public methods reveal an interface to the outside world and might not fail in the context of a program, but can be explored in complex ways.
It might be overkill since design patterns aim to ensure the callers pre-conditions

JCrasher Approach

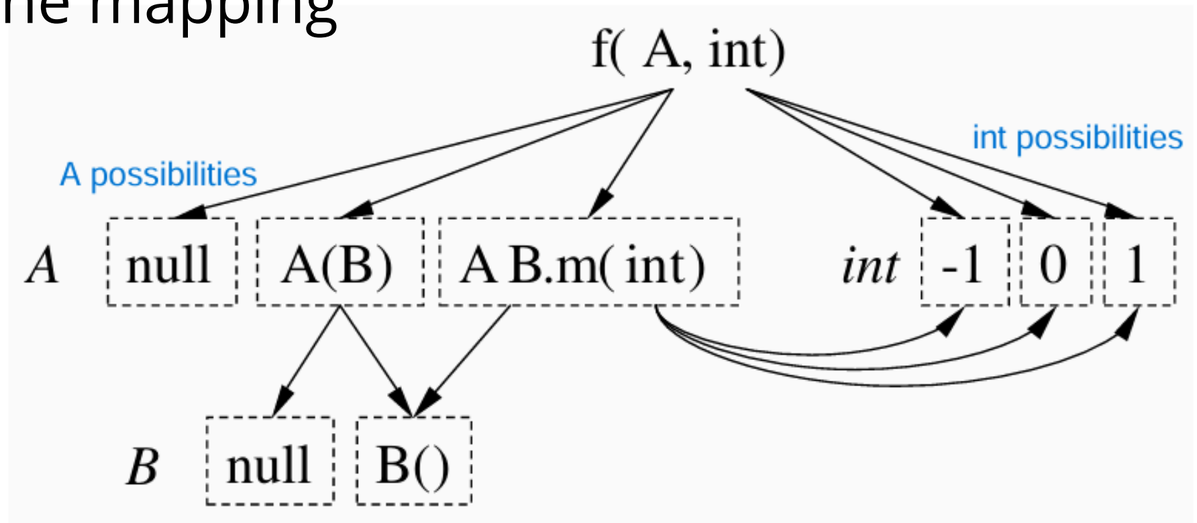
Nevertheless, the developer can choose the code to be tested and under what assumptions.
The JCrasher approach allows to discover inputs that cause errors at a high a deep level, being aligned with good code practices. “Fail Fast”

How JCrasher Works - Overview



Test Case Generation - Detail

- Uses reflection to identify parameters and returning types, and method/variable visibility
- Creates the mapping



- f(null,-1), f(null, 0), f(null, 1), f(new A(null),-1), ..., f(new B().m(1), 1)

Parameter Graph

- JCrasher creates rules like “R <- C, A1, A2, ..., AN”
- Search these rules to create the tests
- Represented as a graph
- Enhances Computation and allows to discover depth of the test

Test Case Selection

- It is possible to compute the size of the parameter-space without creating all possible tests
- Important when because of time constraints
- The user typically knows how much time has for testing
- With this JCrasher calculates how many tests can execute randomly

How are tests structured?

- Base structure similar to standard Java tests
- Methods / constructors invoked inside of a try catch block
- Heuristics define whether or not exceptions should be propagated upstream to JUnit

```
public void test() throws Throwable {
    try {
        // invoke methods / constructors
    } catch (Exception e) {
        dispatchException(e);
    }
}
```

Throwables

Errors

- Typically indicate a system-level problem
- Dispatched to JUnit

Throwables

Errors

- Typically indicate a system-level problem
- Dispatched to JUnit

Checked Exceptions

- Declared
- Not dispatched to JUnit

Throwables

Errors

- Typically indicate a system-level problem
- Dispatched to JUnit

Checked Exceptions

- Declared
- Not dispatched to JUnit

Unchecked Exceptions

- Can be thrown regardless of being explicitly declared
- Subtypes of RuntimeException

Unchecked Exceptions

JVM-triggered Exceptions

- Exceptions from java.lang
  - ArrayIndexOutOfBoundsException
  - ClassCastException
- Dispatched to JUnit

```
public void doWork() {
    Object x = new Integer( value: 0);
    String s = (String) x;
}
```

Precondition violation Exception

- Exceptions from java.lang
  - IllegalArgumentException
  - IllegalStateException
- Can be dispatched to JUnit

```
public void setWeight(int weight) throws InvalidOperationException {
    if (weight <= 0 || weight >= 16) {
        throw new InvalidOperationException("Weight must be between 1 and 15, inclusive.");
    }
    this.weight = weight;
}
```

Resetting Static State

Initial approach

- Single JVM instance
- Problematic - static objects could corrupt the application’s internal state

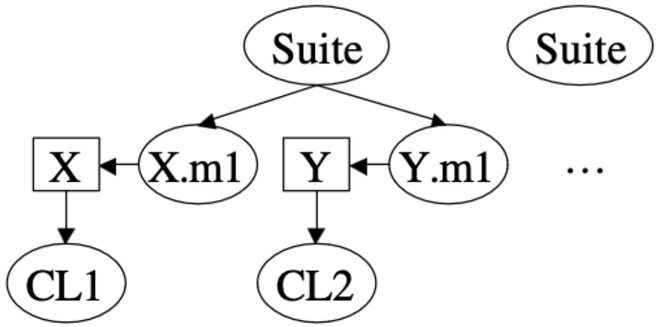
Two possible solutions

- New class loader per test
- Modifying the code under test - at the bytecode level
  - Both require changes to JUnit

Multiple Class Loaders

Modified JUnit – junitMultiCL

1. Each test-case is loaded by a new classloader
2. The hierarchy is split up into a set of suites
3. The garbage collector can free the heap from executed suites



Load-time Re-Initialization of Java Classes

- JUnit’s class loader replaced by a class loader able to re-initialize static components
- No multiple class loaders - less memory

- Not 100% accurate

Performance comparison

Approach	Performance description	Speed comparison	Overhead
Multiple Class Loaders (1)	Fast for reloading a few classes	More than twice as fast as (3)	Significant for larger batches
Re-initialization (2)	Extremely fast	Over 20 times faster than (1)	Negligible compared to other overheads
Restarting the JVM (3)	Slow	Baseline (reference)	High overhead for each test run

Alternatives and Future Directions

Data-driven Re-initialization

Instead of rerunning initialization code, save and restore all static state, but this approach requires sophisticated handling of class loading.

Class Loader Approach

Use multiple class loaders for re-initializing static state in failing tests, balancing correctness and performance.

Static Analysis for Dependencies

Utilize a static analysis algorithm to determine class dependencies for re-initialization, potentially catching bad circular dependencies.

Offline Class Rewriting

Rewrite classes in a separate compile phase so that they are re-initializable, though this assumes all necessary classes are known in advance.

Simulating JVM Initialization

Simulate JVM class initialization exactly by extensively modifying bytecode, but this introduces significant complexity and overhead for minimal benefit.

JCrasher in Practice

Practical Considerations

Two major modes of using JCrasher

Some experiments and results

Types of problems that JCrasher finds, the problems it does not find, its false positives, as well as performance metrics on the test cases.

Major modes

Interactive Mode

Interactive mode allows users to generate simple test files with minimal effort using a plug-in for the Eclipse IDE, making it ideal for quickly identifying shallow bugs in newly developed code.

Batch Mode

Batch mode tests a set of classes by systematically exploring the parameter space and executing numerous tests within a specified time, efficiently identifying issues through automated, large-scale testing.

Experiments & Results

Tables I and II give program and testing performance metrics for the testees.

Let’s check the testees first!

Testees

Raytrace Benchmark

“We ran JCrasher on the Raytracer benchmark of the SPEC JVM suite.
This experiment was not primarily intended to find errors since the application is very mature.
Instead, we wanted to show the number and size of test cases (see Tables I and II) generated for a class in a realistic application, where a lot of methods can be used to create type-correct data.”

Raytrace Benchmark

Code

The constructor below throws a NegativeArraySizeException when passed parameters (-1, 1)

```
public Canvas(int width, int height) {
    Width = width;
    Height = height;
    if (Width < 0 || Height < 0) {
        System.err.print("Invalid window size!" + "\n");
    }
    Pixels = new int[Width * Height];
}
```

Test Case Generation

Generated a test that calls the Write method which in turn calls the SetRed method, leading to an ArrayIndexOutOfBoundsException because of the -1 and 0 values passed to Write.

```
public void test93() throws Throwable {
    try {
        Color c4 = new Color(-1.0f, -1.0f, -1.0f);
        Canvas c5 = new Canvas(-1, -1);
        c5.Write(-1.0f, -1, 0, c4);
    } // [..]
}
```

```
public void Write(float brightness, int x, int y, Color color) {
  color.Scale(brightness);
  float max = color.FindMax();
  if (max > 1.0f) color.Scale(1.0f / max);
    SetRed(x, y, color.GetRed() * 255); //[..]
}
private void SetRed(int x, int y, float component) {
   int index = y * Width + x;
   Pixels[index] = Pixels[index] & 0x00FFFF00 | ((int) component);
}
```

Student Homeworks

Apply JCrasher to test homework submissions from a second-semester Java class, where beginning programmers are likely to introduce errors.

Student Homeworks

```
public static int findPattern(int[] list, int[] pattern) {
    int place = -1; int iPattern = 0; int iList;
    for (iList = 0; iList < list.length; iList++)
        if (isTheSame(list[iList], pattern[iPattern]) == true)
            for (iPattern = 0;
                ((iPattern <= pattern.length)
                && (isTheSame(list[iList], pattern[iPattern]) == true));
                iPattern++)
            { place = iList + 1;
              isTheSame(list[iList + 1],pattern[iPattern + 1]);
            }
}
```

Pattern Matching

Passing ([0], [0]) to the following method findPattern by programmer s1 causes an ArrayIndexOutOfBoundsException, although the input is perfectly valid.

Example Scenario Causing the Exception:

- Given list = [0] and pattern = [0]:
  - The outer loop starts with iList = 0.
  - isTheSame(list[0], pattern[0]) is true.
  - The inner loop starts with iPattern = 0.
  - The condition iPattern <= pattern.length (which is 1 <= 1) is true, so it enters the loop.
  - iPattern increments to 1, but then isTheSame(list[0], pattern[1]) is accessed, causing an ArrayIndexOutOfBoundsException because pattern[1] does not exist.

Student Homeworks

```
public static void main(String[] argv) {
    int tests = Integer.parseInt(argv[0]); //[..]
}
```

Main Method

NumberFormatException when passing String " " to main method.

```
public static int[] getSquaresArray(int length) {
    int[] emptyArray = new int [length]; // [..]
}
```

Negative Array Size

NegativeArraySizeException when passing -1 to getSquaresArray.

```
public BSTNode(Object dat){
    setData(dat); //[..]
}
public void setData(Object dat){
    data = (Comparable) dat;
}
```

Comparable Interface

ClassCastException due to improper type checking.

UB-Stack

Test JCrasher with UB-Stack, previously used as a case study in the testing literature.

UB-Stack

Outcome: JCrasher did not report any problems for UB-Stack.

Results of using JCrasher on real programs (testees)

Class name	Testee Author	Public methods	Test Cases	Crashes	Tests Problem reports	Redundant reports	Bugs
Canvas	SPEC	6	14382	12667	3	0	1?
P1	s1	16	104	8	3	0	1 (2?)
P1	s1139	15	95	27	4	0	0
P1	s2120	16	239	44	3	0	0
P1	s3426	18	116	26	4	0	1
P1	s8007	15	95	22	2	0	1
BSTree	s2251	24	2000	941	4	2	1
UB-Stack	Stotts	11	16	0	0	0	0

Table I

The execution time and disk space required for generated test cases.

Testee		Tests				
Class name	Author	Test Cases	Creation time [s]	Source size [kB]	Execution time [s]	Report size [kB]
Canvas	SPEC	14382	5.0	6000	14.9	30
P1	s1	104	0.3	20	1.0	1
P1	s1139	95	0.3	19	0.7	2
P1	s2120	239	0.3	55	1.3	2
P1	s3426	116	0.3	23	0.6	2
P1	s8007	95	0.3	19	0.5	1
BSTree	s2251	2000	0.9	564	3.4	6
UB-Stack	Stotts	16	0.3	4	0.5	0

Table II

Related Work

JUnit Test Class Generators

- Enhanced JUnit, that just like JCrasher, automatically generates JUnit test classes from bytecode and chains constructors up to a user defined maximal depth.
- JTest
- JUnit Test Generator

JUnit Test Class Generators

“Nevertheless, **Enhanced JUnit** does not attempt to find as many T-generating functions as possible nor does it automatically produce a huge number of combinations as we do. It also does not use subtyping to provide instances.”

```
junit.samples.money.Money implements IMoney {
  /* details omitted */
  public IMoney add(IMoney m) { /* implementation omitted */ }
}
```

Enhanced JUnit will generate the code

```
myMoney.add(new junit.samples.money.IMoney());
```

which will not even compile.

JUnit Test Class Generators

“The commercial tool **Jtest** has an automatic white-box testing mode that works similarly to JCrasher. Jtest generates chains of values, constructors and methods in an effort to cause runtime exceptions, just like JCrasher. Jtest does not seem to attempt to generate a huge number of test cases randomly and consequently it does not deal with the scalability issues that JCrasher has identified. Some of the ideas in JCrasher — fast resetting of static state, for example — should be applicable to Jtest.“

JUnit Test Class Generators

“The **JUnit Test Generator** creates suitable skeletal JUnit test files given java class files. For each method under test, a test code block is generated. All needed instances are initialized with null, however.”

More related work

Providing Random Input

The Ballista project at Carnegie Mellon University has led to automatically crashing different software systems like operating systems.

The “Fuzz Testing of Application Reliability” project at the University of Wisconsin at Madison has done black box testing using random input.

Claessen and Hughes test pure functions in the Haskell programming language using random input.

....

Formal Specifications

Writing a formal specification that is sufficiently descriptive to capture interesting properties for a large piece of software.

A recent piece of work on generating test inputs automatically from specifications in the Java setting is the Korat system.

...

“Instead of expecting formal specifications for deciding what is well-formed input, we rely entirely on type system information.”

Conclusion