Vince Protani

CS 483: Software Testing

Dr. Raunak

11 December 2017

Software Testing Tool Analysis: JCrasher

Overview and Experience

JCrasher is a an automated unit test generating tool using random testing developed in the early 2000s by professors Cristoph Csallner and Yannis Smaragdakis made for testing Java programs. The developers created the tool to produce a easy-to-use program that ideally covers a sufficiently large area of an input domain by using random selection rather than exhaustive testing. They admit that although random testing can be unreliable since there is no sensible method to deciding what inputs to use, it has the advantage of being completely automated. Though not explicitly stated by the developers, the program is a research tool. The developers boast of at least six instances of researchers implementing the tool at universities as well as Microsoft Research. Mentioned as well are two instances of the tool being integrated in an undergraduate course in software verification and validation in addition to a graduate course in program verification. Aside from these examples cited by the developers, I was unable to find any contemporary uses of the program other than infrequent mentions of its name on StackExchange. Even though the tool has not been modified or updated since 2007, it seems to still serve some purpose of exposing bugs in programs written in as late as Java 4.

JCrasher performs by inspecting byte code provided by the user and generating a set of JUnit unit tests to be run on the software under test (SUT). The program can either be run as an Ant program or as a plugin to Eclipse. I chose to run it as an Ant program, which requires working with the terminal and manipulating Ant build files and code within the programs directories. When using Ant, JCrasher must run its “jcrasher.xml” build file, given a file indicating the SUT. It compiles the source code and identifies the parameter input domains for each given piece of the SUT; with this information JCrasher creates JUnit test cases for each class in the program, automatically runs them against the code, and generates a report identifying the errors uncovered. Important to note is that JCrasher has the ability to distinguish between exceptions that indicate faults and exceptions that are expected, which is taken into account when reporting errors. Files are generated and stored in the program’s directory containing the source code for each unit test as well as the report of errors.

To get started with JCrasher, the developers provide fairly ample information on a Google Code webpage about how the program works with Ant along with a description of dependencies (JVM, JUnit 3.8.1, etc.), download links, and a handful of discovered bugs. However, since this program has not been touched in years, the page is outdated. Almost all of the links provided are no longer usable, the download links do not actually provide the program as stated in its description (though I discovered the correct .zip file made available by third-party providers on GitHub), and some of the issues identified with the program are not even addressed though stated. The use of older versions of JCrasher, including the Eclipse plugin, are also explained on another page linked on the program’s homepage, but these are virtually useless now that the program is so old. For example, the Eclipse plugin has well-explained steps to install and use it, but the plugin does not work in any easily obtainable version of Eclipse or Java today.

My personal experience with the program was frustrating to say the least. I spent over 15 hours trying to configure the program so that it would work on Windows machines, but it did not. I consistently got errors about multiple programs using features incompatible with Java 4, the JUnit unit tests not being able to compile, and the main class (which was supposed to run the test cases) not being able to be found, among other issues pertaining to configuring Ant and editing the build files which had a number of errors. The developers even updated the build files under the identified issues section of the Google Code page hoping to fix the compilation error, but the new build files did not work either. I finally decided to try running the program on a Linux machine and it worked without fail. Though the program worked on the examples provided, I could not select a program of my own or of interest to me that was actually compatible with JCrasher. I imagine that if the program were to be updated to be compatible with more recent versions of Java and Eclipse, it would be an indispensable tool in beginning testing since the test cases are automated; this way, some bugs can be exposed early without having to spend time considering the input domain of a program and carefully selecting test cases.

Software under Test

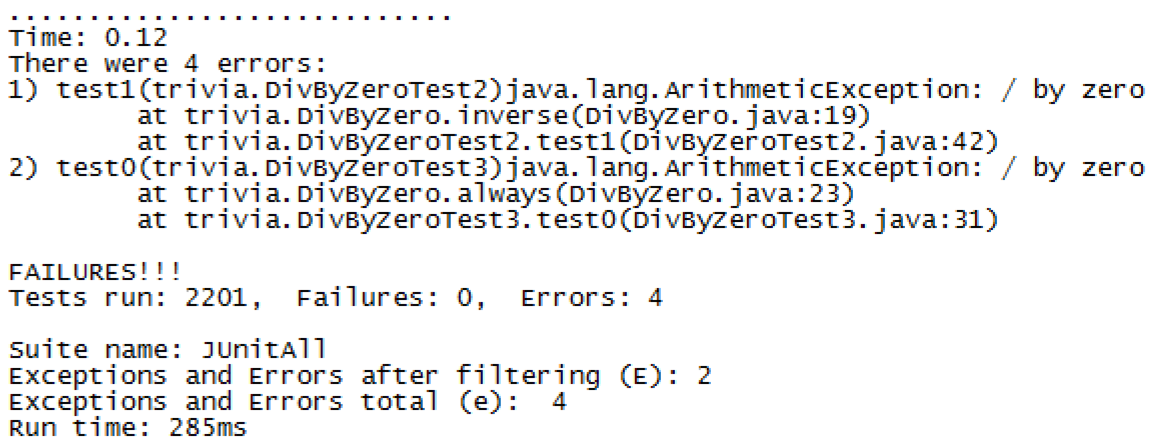
Initially, I tried to test the Hangman program I wrote as a freshman since it has plenty of user interaction and a fairly large input domain, but Java 4 could not support the use of generics, which abound in the Hangman Program. I then tried to use a program that uses calculus to compute derivatives of given functions since the program has about ten classes, the input domain is large, and I am unashamedly a big fan of calculus. However, similar to Hangman, the program implements features unavailable in Java 4. JCrasher's inability to work with any contemporary programs limited my choice of programs written by myself that had any sufficient complexity. Further, I struggled to find any programs available on sources like Github that were relatively complex and interesting but that would also meet JCrasher's requirements. Therefore, left with few other options, I decided to test JCrasher with the example programs provided. Though they lack complexity, they were guaranteed to be compatible since they were written at the same time as the program was in use.

There are three programs containing one class each that were under test: DivByZero, ManyParameters, and NullDeref. DivByZero is just 25 lines of code and takes as input an integer i. There are two methods: one which divides 1 by i and another which divides i by 0. ManyParameters is 19 lines of code that accepts seven integers as input and returns the sum of them. NullDeref takes as an argument an Object and returns a reference to it. Though these programs are basic, their output is easy to anticipate. For example, it is clear to see DivByZero will fail when the input is 0 and NullDeref will fail when the input is null, so these should ideally show in the test cases run.

Evaluation

When inspecting this tool I am looking for some specific characteristics, such as randomness of input, number of test cases, and exposure of anticipated failures. One benefit of JCrasher is the immediacy of unit test generation. This saves a load of time for programs with large input domains since many test cases can be produced in a little amount of time. JCrasher's ability to distinguish between expected exceptions and unexpected exceptions is beneficial as well. These benefits, however, only pay off if the input is representative of the input domain, i.e. input is random and pulls from the entire domain; the number of test cases is large; and bugs I already know of are exposed through the tests. Truly random inputs should pull from all of the domain, which gives a better chance of hitting error-prone cases. Similarly, though it does not guarantee bugs are exposed, having more test cases increases the likelihood of bugs being found. Ultimately, if the program finds bugs I am expecting, it has done a good job within less time than I would have creating test cases of my own.

Errors that I expected to encounter were i = 0 in DivByZero, numbers whose sum would exceed the size limit of an int in ManyParameters, and a null reference passed into NullDeref. After running the program, the first thing I checked was the report file with errors found.



```
............................
Time: 0.12
There were 4 errors:
1) test1(trivia.DivByZeroTest2)java.lang.ArithmeticException: / by zero
        at trivia.DivByZero.inverse(DivByZero.java:19)
        at trivia.DivByZeroTest2.test1(DivByZeroTest2.java:42)
2) test0(trivia.DivByZeroTest3)java.lang.ArithmeticException: / by zero
        at trivia.DivByZero.always(DivByZero.java:23)
        at trivia.DivByZeroTest3.test0(DivByZeroTest3.java:31)

FAILURES!!!
Tests run: 2201,  Failures: 0,  Errors: 4

Suite name: JUnitAll
Exceptions and Errors after filtering (E): 2
Exceptions and Errors total (e):  4
Run time: 285ms
```

JCrasher output file after running all test cases on DivByZero, ManyParameters, and NullDeref

This was initially disappointing being the result of all test cases run on each of the three classes. Between the three, only four errors in total were caught (only two when considering expected errors) and each was in DivByZero. In fact, the same error (i = 0) was caught by two different test cases, so the only error caught was expected. This means that the errors expected in the other two classes were not caught, which prompted me to look through the unit tests and get an idea of what exactly was being tested.

JCrasher created three unit test classes for DivByZero, six for ManyParameters, and two for NullDeref. The first of each only tests the constructor and the rest test inputs. The DivByZero class tests only used -1, 0, and 1 as inputs, and so did not use much at all of the input domain. This is probably sufficient though considering it tests a negative number, a positive number, and 0. The inputs for ManyParameters were more disappointing because it only tested with variations of -1, 0, and 1 as well. This seemed like more of a combinatorial approach with very few possibilities for each input variable than random testing. Every number other than those three - especially large numbers - was ignored, which allowed the overflow error to pass undetected. As for NullDeref, both instantiated objects of multiple types and null were tested, but for some reason the null reference was not picked up as an error.

Overall, JCrasher in my opinion did not do a good job of testing. The input was not random especially for inputs of integer values being only -1, 0, and 1. If the inputs were truly random, there would have been a greater span of numbers tested and there would be a better chance of catching an overflow issue in ManyParameters. Furthermore, for being such a restrictive testing domain, there was an extreme number of test cases specifically for ManyParameters - each unit test class had a hundred or several hundred tests. A human tester could have identified more useful test cases while using fewer. A combinatorial testing approach could have covered the same variations of -1, 0, and 1 with fewer test cases. Even with so many test cases, expected bugs were not found. Considering my criterion and an alternative test method, such as writing unit tests, JCrasher does not perform well.

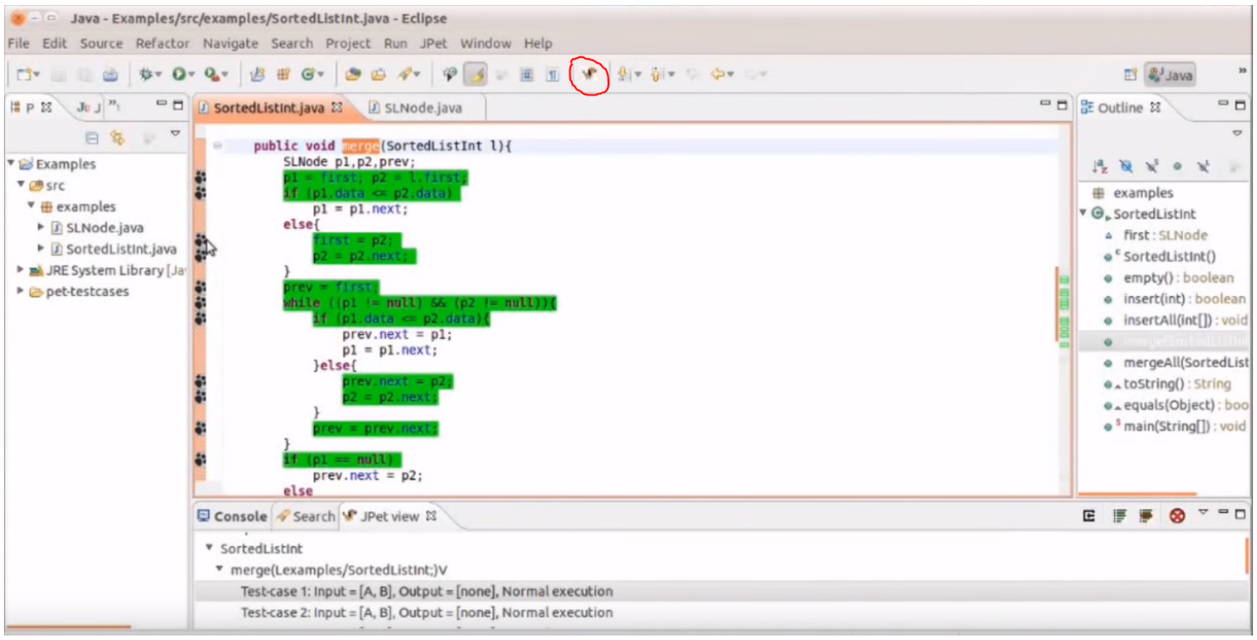
Testing Methodology

Random testing paired with unit testing as employed by JCrasher is a black box testing approach. As the developers state, the program does not scan the source code, rather it reads the compiled program files to gather the input space. Though it has knowledge of method parameters and names, it has no understanding of how the program operates. These unit tests are written solely with the knowledge of what methods expect as parameters.

This type of testing is most applicable to any programs as a beginning approach to testing. Although JCrasher's input does not seem random as I have said, if it were random, a program such as this has a decent chance of exposing bugs early on without the effort of creating unit tests manually. A random testing approach may also be beneficial for programs that expect user input, like the VendingMachine, since random input could mirror what a user of the program may put in whether purposefully or accidentally. This can be a quick procedure done to expose minor bugs before going onto more considerate testing methods, such as coverage criteria or data flow analysis.

Comparison of Similar Programs

In the realm of automated test generation, the best alternative to JCrasher I found was jPET. This program is an Eclipse plugin that generates unit tests for code selected by the user and additionally considers specifications given by the user, such as the domain of integer values to test. Firstly, the interface is much more user-friendly than JCrasher; it adds a button onto the Eclipse taskbar that when clicked leads to a menu of specifications for generating a unit test of the code open in the Eclipse window. The tests are then run and output is shown on the console. Each individual unit test can be observed for what inputs were and what corresponding output was. jPET even shows the type of coverage the user specifies as seen below.



```
public void merge(SortedListInt l){
    SLNode p1,p2,prev;
    p1 = first; p2 = l.first;
    if (p1.data <= p2.data)
        p1 = p1.next;
    else{
        first = p2;
        p2 = p2.next;
    }
    prev = first;
    while ((p1 != null) && (p2 != null)){
        if (p1.data <= p2.data){
            prev.next = p1;
            p1 = p1.next;
        }else{
            prev.next = p2;
            p2 = p2.next;
        }
        prev = prev.next;
    }
    if (p1 == null)
        prev.next = p2;
    else
```

```
SortedListInt
  merge(Lexamples/SortedListInt;)V
    Test-case 1: Input = [A, B], Output = [none], Normal execution
    Test-case 2: Input = [A, B], Output = [none], Normal execution
```

jPET demonstration: https://www.youtube.com/watch?v=fNIWmxdyW6Y

Relative to JCrasher, jPET is more contemporary, more understandable, and easier to use. It also offers more features, such as the ability to define integer input spaces and request coverage criteria. jPET is also easier to install, only requiring a link to be used within Eclipse to install new software.

Similar to JCrasher is another command-line automatic unit test case generator called Randoop. Immediately upon discovery, I felt Randoop was a better program than JCrasher because it is contemporary (able to use easily in 2017) and its webpage has ample directions on installation and use that are well-explained and well-organized with specific examples from running the program. The program also simplifies the process by only running on Java rather than being dependent on another program like Ant. Additionally, Randoop offers specifications such as input limits like jPET. The one con of using this program is that it seems to operate more slowly, but it provides more reliable input and is applicable to more recent versions of Java. However, this program is more sophisticated, allowing the user to test more simply and more accurately.

Compared to other automatic unit test case generators, JCrash fails mainly because of its incompatibility and outdatedness. If it were to be updated to be compatible with newer versions of Java and Eclipse and to have specifications of input boundaries, it would be comparable to these other tools.

Conclusion

After installing and testing JCrasher, I would not suggest this tool to others, whether for research or for personal or commercial use. The program is no longer feasible since it has not been updated in about ten years; it is incompatible with any program that one would be writing nowadays. Even if it were, compared to alternatives JCrasher offers nothing in terms of input specifications or user-friendliness. The program was only relatively easy to use on a Linux machine and impossible to use on a Windows machine after hours of trying. Even when I was finally able to work the program, it was overwhelming to work in a directory that has so many documents, subfolders, build files, and dependencies. The program could be a lot more user-friendly by streamlining its processes into a .jar file and a .xml file, for example.

Though I believe random testing and automatic unit test case generation is applicable in the beginnings of testing a program, I would not recommend this specific program to anyone. The alternatives offer more flexibility in how classes are tested and higher-level, cleaner interfaces. If it were to be a useable program, JCrasher would have to be thoroughly updated to meet the standard competing automatic test generators have created.