JCrasher: an automatic robustness tester for Java

A report as part of the seminar Software testing

Theo Harkenbusch

Paderborn University, Institute for Computer Science,
33102 Paderborn, Germany
theohark@mail.upb.de

Abstract. Software testing has been around for over 40 years now. It is the most common way to uncover defects in a program. If a program behaves correctly even under invalid inputs, it is said to be *robust*. Program code can be tested for this property with the help of *robustness testing*. This paper presents an accessible overview of a particular robustness testing tool called *JCrasher*, which produces test cases for the popular Java testing framework JUnit. JCrasher is designed to automatically test the robustness of Java code. It tries to make the program under test throw undeclared runtime exceptions, causing it to terminate forcefully. This is done by analysing the type information of public methods and constructing calls to those methods providing random but syntactically valid data as parameters. To construct these well-typed parameter combinations a special data structure is used. JCrasher also makes use of novel heuristics to determine whether an exception it caused is the result of a bug or if it indicates the violation of some specified precondition. Furthermore, it also includes a unique state-resetting mechanism to ensure that tests do not run on states modified by previous tests.

Keywords: Software Testing, Test Case Generation, Random Testing, Java, State Re-initialization

1 Introduction

On the 4$^{\text{th}}$ of June 1996, the maiden flight of the European Space Agency's new Ariane 5 rocket ended in an expensive disaster. After only 37 seconds into the mission, the rocket exploded midair [7]. Although no one was hurt during the incident, scientific equipment valued at around 500 million dollars was destroyed [8]. After a thorough investigation, the inquiry board reported a software exception as cause of the incident. During the conversion of a floating point number, "a value greater than what could be represented by a 16-bit signed integer" was assigned to a 16-bit integer [7]. The corresponding program code was not able to cope with an internal function call returning an unexpected high value, thus resulting in the crash of the rocket. Although the method that caused the exception could have been guarded against such a condition, it was not. The Ariane

5 incident can be seen as an extremely costly example for the demand of *robust* software.

Software robustness can be defined as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" according to the IEEE standard glossary of software engineering terminology [6]. With *JCrasher*, Csallner and Smaragdakis propose a way to test the (software) robustness of Java applications automatically without the need of supervision [3]. To achieve this level of automation, a random testing approach is incorporated. JCrasher produces random inputs which are supplied to public methods of the program under test. As JCrasher tries to get the program under test to crash by throwing an undeclared runtime exception, it is crucial that the randomly generated data is also type-correct in terms of Java semantics as otherwise its generated test cases could not be compiled. To ensure the construction of correctly typed inputs, a special data structure is utilized (Section 2.2). The authors note that JCrasher has been designed with practicality in mind as it outputs test cases for the popular Java testing framework JUnit. These test cases can easily be integrated into a typical industrial development cycle where an incremental built software project is tested during night time [3].

Csallner and Smaragdakis are, however, not the first to propose such an automatic testing approach based on the principles of random testing (Section 3). Nevertheless, JCrasher offers some novel concepts that are noteworthy. For instance, it incorporates a unique heuristic function which helps to decide if a caught Java exception was raised because of a genuine bug or if it is merely the result of a precondition violation (Section 2.3). Csallner and Smaragdakis also propose a new technique to ensure that previously run test cases will not affect subsequent test cases. This is done with the help of a special re-initialization strategy for static data which will be discussed later on.

The following sections will explain the general functionality of JCrasher in detail. Special emphasis will lay on the proposed novelties. Note that the following concepts are based on the original JCrasher paper by Csallner and Smaragdakis if not stated otherwise.

2 Architecture of JCrasher

JCrasher tests the robustness of Java program code. Hence, it requires a Java program as input—specifically, its bytecode. As output it produces numerous JUnit test cases. It is not uncommon to generate up to two million test cases in a five hour time span, depending on the complexity of the Java program under test [3]. By auto-generating persistently saved JUnit test cases, JCrasher allows for human inspection of the tests. This means that developers can cherry pick certain test cases that are particularly successful and integrate them into their regular testing suite. The next section will give insight on how these test cases are generated.

2.1 Test Case Generation

With the help of Java reflection, JCrasher scans the provided bytecode for methods with the access modifier set to public. The robustness of public methods is something particular interesting to test against, as public methods can be seen as way to interact with the world around the program and are thus influenced by external parameters [3]. JCrasher is looking for values which it can supply to the found public methods as parameters in order to crash the target program. If said method then crashes, i.e. throws an undeclared runtime exception, it follows that it is *not* robust. It would be the responsibility of the callee to enforce possible preconditions regarding the provided parameters [3]. The preferred way to do this in Java is to throw an `IllegalArgumentException` to let the the caller know that the method does not know how to deal with the provided parameters [3].

For each declared method **f** that JCrasher finds, it will produce a random sample of executable test cases. Each of these generated test cases will pass a different parameter combination to **f**. For example if we want to test a method `public void setAge(int age)`, we could construct numerous test cases and provide the method with different integer values. In theory we could construct 2^{32} possible test cases for this method alone, as this is the space of possible integer values in the Java programming language [5]. The number of possible tests is, therefore, extremely huge and JCrasher can only test so much of it. However, JCrasher can work autonomously and is, therefore, very cheap to use [3]. It can be used to assist other testing techniques and it using it does not introduce any disadvantages.

Java is a programming language with object-orientated features, therefore, it is possible that a method not only expects primitive data types (`boolean`, `byte`, `char`, `int`, `float`, `double`, `long`, `short` [5]) as input but also Java objects. Consequentially, JCrasher must be able to construct the needed Java objects in order to test found methods. This is why JCrasher keeps track of how to construct objects of different types by using a special data structure: The parameter-graph.

2.2 Parameter-Graph

The parameter-graph is an in-memory data structure that represents the parameter-space of the methods in the input program. It is essentially a mapping from Java types to either methods returning that particular type or to some pre-set values [3]. An example for a pre-set value would be `1`, `0` and `-1` for integer values. Additionally, `null` is a reasonable substitute for any reference type and can be used for any object [3]. JCrasher uses the parameter-graph to decide how many and which test cases it generates for a given method.

Different parameter combinations, i.e. inputs, for a certain method are constructed by traversing the parameter-graph.

Suppose JCrasher wants to test method `m` of class `X`. Further suppose that `m` may take an Java object of type `O1` and returns an object of type `R`. Method

m's signature looks like this: `X.m(O1) returns R`. JCrasher infers from this that it needs to know how to construct an object of type `O1` to successfully test method `m`. Additionally, JCrasher infers from the return type that method `X.m` is a valid way to construct objects of type `R` [3]. This knowledge is kept inside a graph structure.

A benefit of using graph-like structures is, they are rather easy to visualize. Figure 1 illustrates how JCrasher chooses a suitable parameter-combination for a method. The root element of the graph contains the method we want to test, here it is method `f` of class `DemoClass`. An edge in the parameter-graph, from a type to a method or an explicit value, denotes a way to construct a value of this specific type. Multiple edges from the same source simply depict alternative ways to construct values. For instance, the three outgoing edges from `int` mean that JCrasher can select one of the three values `-1`, `0` and `1` as an actual parameter.

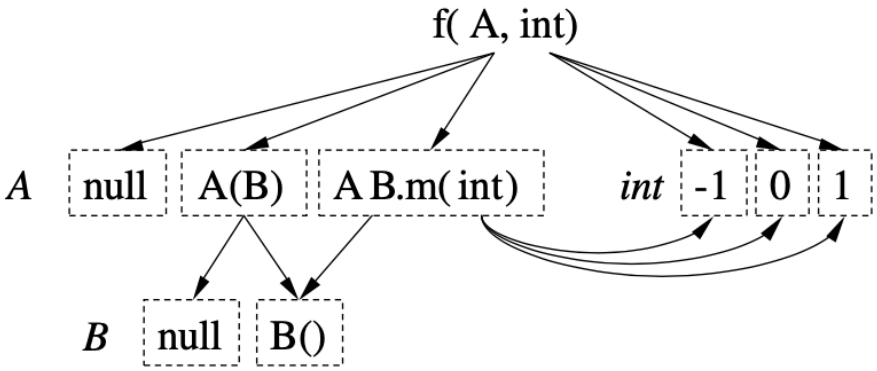

Fig. 1. Traversable parameter-graph to create different parameter combinations [3, p. 1030].

Essentially, traversing the parameter-graph means creating different parameter combinations for a certain method. This particular graph in Figure 1 yields the following possible combinations: `f(null, -1)`, `f(null, 0)`, `f(null, 1)`, `f(new A(null), -1)`, ..., `f(new B().m(1), 1)`.

JCrasher can derive test cases based on these parameter-combinations. Each parameter-combination gets translated into one test case. Figure 2 shows one of the possible test cases which could be generated by traversing the above parameter-graph. First, an object of class `DemoClass` is generated by its default constructor. Afterwards, the method under test, `f`, is called. Note that, every single test case is wrapped in a `try-catch` block.

```
public void test1() {
  try {
    //test case
    DemoClass c = new DemoClass();
    c.f(new A(null), -1);
  }
  catch (Exception e) {
    dispatchException(e);
  }
}
```

Fig. 2. Derived test case from the parameter-graph in Figure 1. The actual call to the method along with the parameter-combination `f(new A(null), -1)` [3, p. 1032].

These test cases can be executed with JUnit. During the execution, JUnit catches every possible Java `Exception` that the method under test may throw. It simply dispatches the caught exceptions forward to the JCrasher runtime. Based on its novel heuristic functions, JCrasher will decide if the exception may indicates a bug in the method under test or is possibly caused by a precondition violation. The exception is then reported back to JUnit (which flags the test case as failed) or suppressed, respectively [3]. The following section will describe the heuristics which these decisions are based upon.

2.3 Heuristic Approach

JCrasher uses a heuristic approach to differentiate whether a caught exception during the test execution was caused by a genuine bug or merely by violating possible preconditions of the method under test. The reason is that JCrasher wants to reduce it's number of reported false positives, i.e. test cases that are falsely flagged as failing when in reality no robustness problem is present. The approach is based upon the Java sub-class hierarchy of `java.lang.Throwable` [3]. The main idea is to use the class which the given exception extends as a guiding rule. Figure 3 illustrates this by annotating the class-hierarchy of `Throwable` with recommendations how to deal with this kind of exception.

An `Error` always denotes a serious problem, e.g. when the JVM (Java Virtual Machine) runs out of memory space. Thus, errors should never be caught. And, indeed, the generated JUnit test cases only catch `Exceptions` and never an `Error` as seen in Figure 2.

Exceptions can be divided into two groups in Java. The first group being so called *checked exceptions*. These exceptions have to be declared by a special `throws` statement in the method's signature. The most prominent example is the basic `IOException` which indicates problems with the input or output; it often occurs when (miss-)handling files. If JCrasher encounters a checked exception, it assumes that it was thrown to enforce some kind of precondition. Checked

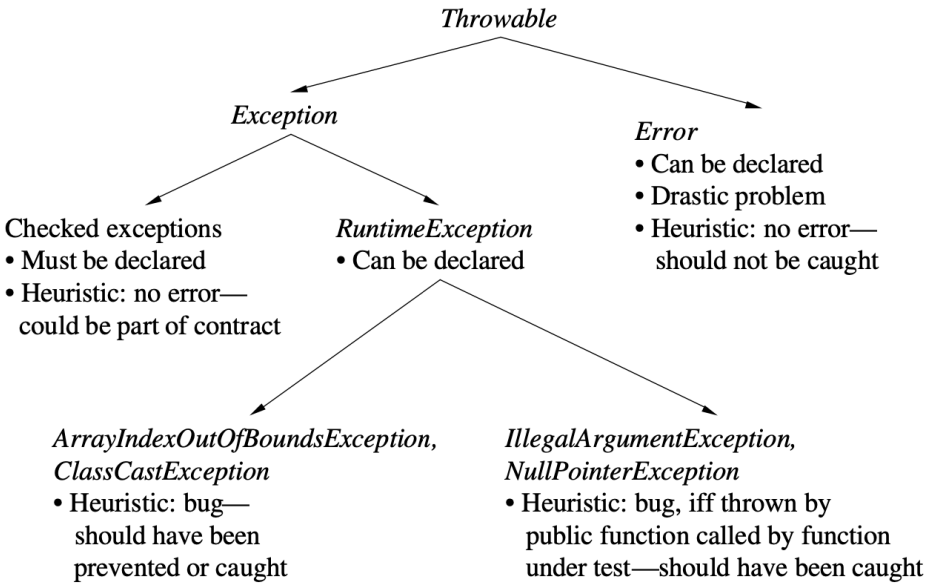

Fig. 3. The Java sub-class hierarchy of `java.lang.Throwable` is used to determine if a caught exception was raised by a robustness bug [3, p. 1033].

exception are therefore never reported back to JUnit. They are suppressed and not considered to be caused by a robustness bug. Figure 4 illustrates this with a fictive method that would throw a checked exception if the given integer parameter was less than 10.

```
// testing with i = 5;
public void method(int i) throws CustomException {
  if (i < 10)
    throw new CustomException();
  // ...
}
```

Fig. 4. Example of a non-bug according to the heuristics as the method under test would throw a checked exception of type `CustomException` when provided with parameter `i = 5`.

The second group consists of *unchecked exceptions* which are *runtime exceptions*. In contrast to checked exceptions, runtime exceptions *can* be declared by methods or constructors but a declaration is not obligatory. Any method may potentially throw an unchecked exception [3]. Unchecked exceptions can again be subdivided into two groups (see the bottom of Figure 3). The general

idea is that unchecked exceptions of the first group are "typically thrown by low-level functions mostly implemented by the JVM" while the second group indicates a violation in regards to some precondition. Examples of the first group of unchecked exceptions are `ArrayIndexOutOfBounceException` and `ArithmeticException`. If an unchecked exception of this group is dispatched to JCrasher, it will be reported to JUnit as a potential robustness problem. Even if the tested method `f` did not throw the exception directly but called another method which then threw the exception, `f` is to blame for not catching and handling it [3]. Consider Figure 5 for an example of what JCrasher considers a robustness problem.

```
// testing with pos = 0;
public void method(int pos) {
  int[] myArray = {2, 4, 8};
  // ...
  myArray[pos];  // ArrayIndexOutOfBoundsException
}
```

Fig. 5. Example of a bug according to the heuristics as the method under test would thrown an unchecked `ArrayIndexOutofBoundsException` when provided with parameter `pos = 0`.

However, if JCrasher encounters an unchecked exception of the second group, it usually indicates that the used parameters violate the method's precondition. Examples are `IllegalArgumentException` and `IllegalStateException` which are both considered Java best practise for reporting problems with the provided inputs [3]. Consequentially, JCrasher's heuristics tend to suppress these exceptions as they are usually not caused by program bug. Please note that there were a few special cases mentioned in the original JCrasher paper that depend on an analysis of the call-stack [3]. Due to the restricted nature of this report, we will not discuss them here.

2.4 Static State Resetting

When multiple test cases are executed sequentially, there is a chance of unwanted interdependencies between the test cases. All test cases that are executed in one single JVM instance share the same class object at runtime (see Figure 6). If one of the test cases now modifies the static state of this shared class object, all other instances of this class that run in the same JVM reflect this modification. This is highly undesirable since the reason of a failing test case can not be determined precisely any more [3]. A preceding test case might have tinkered the static state which in turn caused the test case to fail even in the absence of a robustness critical code.

There are three possible solutions to this problem.

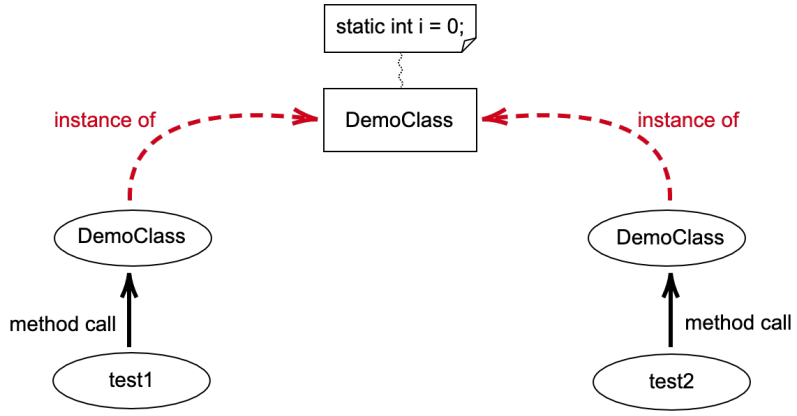

Fig. 6. All test cases use the same class object at runtime as there is only a single JVM instance.

The most naive solution would be to simply use multiple JVMs in order to avoid the interdependencies. However, this approach involves the expensive task of starting, loading and tearing down each individual JVM instance which is why the other approaches seem more reasonable [3].

The second approach is to let each test case operate on a separate copy of a class object by using different class loaders. This way no test case will modify the static state of a class object another test can access.

Nevertheless, the solution that JCrasher actually makes use of, is the third one: "Modifying the code under test at the bytecode level to expose static re-initialization routines, which get called after the end of each test case" [3].

They conducted multiple experiments on the performance of each proposed solutions and found out that use of multiple class loaders is approximately twice as fast as the naive approach. However, the class loading approach was about 20 times slower compared to the re-initialization [3]. For this very reason, the following paragraph will only concentrate on the technicalities of the re-initialization technique.

The idea is to imitate the original JVM initialization progress and re-initialize already used classes after each test execution [3]. For this to work, the loaded classes are manipulated at bytecode level. In Java the method `<clinit>()` contains the compiled variable initializer for static fields of a class. This method gets called by the JVM only at the beginning when the class needs to be loaded. It can not be called from regular Java code as the method name contains square brackets, which are invalid characters. The approach replaces the JUnit class loader with a custom one. Essentially, the idea is to create a copy of that method which is callable by custom Java code. The original static initializer method is then modified to call the copy and afterwards notify JCrasher to add this class to a list of classes that are flagged for re-initialization [3]. After each test case

execution, JCrasher calls the copied method for each class in that list again. This reset the static variables of the loaded classes back to the value they would have been when they were originally initialized.

It is noteworthy that this approach, although really fast, differs from the original Java initialization. The order of the class re-initialization depends on the order of the original initialization. This can potentially lead to some problems especially if there happens to be a cyclic dependency between static data of different classes. However, Csallner and Smaragdakis note that the odds of this happening are highly unlikely [3].

3 Related Work

Software testing has been around for a long time now. It became an essential part in the work routine of software engineers. Therefore, it exists a vast amount of different ideas and techniques to improve the testing process. In the original JCrasher paper some work that is most similar to JCrasher has already been discussed. This section will therefore briefly present some newer testing techniques that somehow are related to the approach presented in this report instead.

Randoop ("**rand**om tester for **o**bject-**o**rientated **p**rograms") is a tool that generates test cases for Java programs by using a so-called feedback-directed random testing approach [9]. This approach is heavily motivated by random testing. However, it enhances the normal random testing approach by using feedback it gathers from executing tests with freshly generated inputs in order to avoid producing illegal and redundant inputs [10]. When a new possible input for a method is generated, it is automatically executed and checked against optionally user-provided contracts. Randoop will then output two test suits. The first one includes test cases that violate the provided contracts while the second test suits contains regression tests that do not violate contracts and can be used to discover deviant behaviours of two versions of the same software [9]. By using runtime-feedback Randoop can avoid generating redundant inputs which can occur when following a normal random testing approach. The authors of Randoop claim that they found severe flaws in commonly used commercial software as well as in open-source projects [10].

Palulu is another tool aimed to simplify software testing. It tries to tackle the problem that input generators often produce more illegal than helpful inputs when not provided with a formal specification. The combination of dynamic analysis and random testing alongside an example execution of the program under test yields a model of legal call sequences. This model influences a random input generator to produce more legal but still diverse inputs [1].

4 Conclusion

JCrasher aims to uncover robustness related bugs in Java programs. It produces JUnit test cases that test public accessible methods by providing it with randomly generated but type correct inputs. JCrasher uses a special graph-like

structure, the parameter-graph, to construct these random and syntactically correct values efficiently after analysing the bytecode of the program under test. If during the execution of these test cases an exception is raised, the exception is evaluated according to a novel heuristic function. Based on the type of the exception JCrasher tries to determine whether the cause of the exception was a (robustness) bug or if the generated test case possibly violated some unknown preconditions. Additionally, JCrasher incorporates a unique static state-resetting mechanism that prevents test cases from introducing unwanted side-effects for subsequent test cases. Since its realease, JCrasher has been gradually improved and the authors presented several follow-up tools in the years after [2] [4].

References

[1] S. Artzi et al. "Finding the needles in the haystack: Generating legal test inputs for object-oriented programs". In: (2006).

[2] C. Csallner and Y. Smaragdakis. "Check'n'crash: combining static checking and testing". In: *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 422–431.

[3] C. Csallner and Y. Smaragdakis. "JCrasher: an automatic robustness tester for Java". In: *Software — Practice & Experience* 34 (11) (2004), pp. 1025–1050.

[4] C. Csallner, Y. Smaragdakis, and T. Xie. "DSD-Crasher: A hybrid analysis tool for bug finding". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17.2 (2008), p. 8.

[5] J. Gosling et al. *The Java language specification*. Addison-Wesley Professional, 2000.

[6] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84.

[7] J.-L. Lions et al. *Ariane 5 flight 501 failure report by the inquiry board*. 1996.

[8] R. A. Maxion and R. T. Olszewski. "Improving software robustness with dependability cases". In: *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224)*. IEEE. 1998, pp. 346–355.

[9] C. Pacheco and M. D. Ernst. "Randoop: feedback-directed random testing for Java". In: *OOPSLA Companion*. 2007, pp. 815–816.

[10] C. Pacheco et al. "Feedback-directed random test generation". In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 75–84.