SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

JCrasher: an automatic robustness tester for Java

Christoph Csallner[1,†] and Yannis Smaragdakis[1,*,‡]

[1] *College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA*

SUMMARY

JCrasher is an automatic robustness testing tool for Java code. JCrasher examines the type information of a set of Java classes and constructs code fragments that will create instances of different types to test the behavior of public methods under random data. JCrasher attempts to detect bugs by causing the program under test to "crash", that is, to throw an undeclared runtime exception. Although in general the random testing approach has many limitations, it also has the advantage of being completely automatic: no supervision is required except for off-line inspection of the test cases that have caused a crash. Compared to other similar commercial and research tools, JCrasher offers several novelties: it transitively analyzes methods, determines the size of each tested method's parameter-space and selects parameter combinations and therefore test cases at random, taking into account the time allocated for testing; it 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; it includes support for efficiently undoing all the state changes introduced by previous tests; it produces test files for JUnit—a popular Java testing tool; and can be integrated in the Eclipse IDE.

KEY WORDS: software testing, test case generation, random testing, Java, state re-initialization

1. Introduction

Testing is the predominant way of discovering program errors. A large volume of research work on testing is based on formal specifications, program mutation, program analysis to derive constraints, white-box coverage-based testing, and more. Nevertheless, in practical settings most testing is either *black box* or static regression testing—both fairly crude but effective techniques. In black box testing [1], a tester interacts with the entire application without any

*Correspondence to: Yannis Smaragdakis, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA
†E-mail: christoph.csallner@cc.gatech.edu
‡E-mail: yannis@cc.gatech.edu

knowledge of its internals. In static regression testing, the developer maintains a suite that exercises different pieces of overall functionality. Every new version of the code is tested against the regression test suite.

Random testing is an alternative to black-box and static regression testing. It consists of providing well-formed but random data as inputs to a program and checking—typically with limited human help—whether the results are correct. In this paper we present a tool, called JCrasher, for random testing of Java classes. JCrasher generates random but type-correct inputs in an attempt to cause a Java application to "crash", that is, to throw an unexpected exception that is not consistent with good Java programming practices for signalling illegal inputs.

One can justifiably have doubts about the efficiency of a random testing approach. The space of possible tests is enormous and "blind" testing can only explore a tiny fraction. Nevertheless, random testing has a few advantages: First, it is very cheap, since it requires no user input except when a potential error is actually found. Second, it can easily cover shallow boundary cases, like integer functions failing on negative numbers or zero, data structure code failing on empty collections, etc. In the case of Java public methods, our testing approach is particularly welcome: although programmers often do not realize it, public methods are an interface to the outside world. This interface can be manipulated in many complex ways and it is often the programmer's responsibility to enforce the preconditions of a public method and report the error—by throwing an `IllegalArgumentException`, for example. JCrasher helps test many "unexpected" scenarios because it looks for ways to combine methods from the available public interfaces in order to create data and state that are type-correct, but potentially erroneous.

JCrasher has been designed with an emphasis on practicality. For example, the scalability goals of the project are defined by taking a usual development cycle into account. As a result, Java developers can easily integrate JCrasher into a typical 24-hour industrial development cycle. Such a cycle includes daily incremental development, building, and testing, followed by a nightly pause, when individual developers import the changes introduced by all other developers in their group, perform a clean build of the project, and run a suite of regression tests against it. JCrasher can be used to supplement such regression tests with automatically generated tests, selected at random to fill the time available for testing. JCrasher could also run unsupervised on a separate machine in parallel with regular development.

Certainly, JCrasher is not the first or only tool of this nature. Several research tools [2, 3, 4, 5, 6] and commercial products [7, 8] employ some of the same ideas in their testing. Nevertheless, JCrasher offers some features that are unique—to different extents compared to different alternatives.

- JCrasher constructs test cases at random, through the tested methods' parameter-spaces taking into account the time allocated for testing. Unlike other tools—Enhanced JUnit [8], for example—JCrasher takes the Java type system into account when constructing random inputs, so that well-formed inputs are constructed by chaining program methods.
- JCrasher 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. The recently introduced assertion facility makes

no difference—it is not clear whether an assertion failure is the responsibility of the current method or of its callers.
- JCrasher ensures that every test runs on a "clean slate": changes to static data by previous tests do not affect the current test. This is done very efficiently within a single Java virtual machine instance, using one of two user-selected approaches: either each test is loaded by a different class loader, or the bytecode of the program under test is re-written at load time to allow re-initialization of static data. We discuss the relative advantages of the two approaches in later sections.
- JCrasher produces test files for JUnit [9]—a popular Java unit testing tool—and can be integrated as a plug-in in the Eclipse IDE. The advantage is dual: First there is significant ease of use and inspection of tests. Second, test plans are reified in code, so that if a developer decides that a test is good enough, he/she can permanently integrate it in a regression test suite.

In the rest of the paper, we will explain the entire structure of the JCrasher system *paying particular emphasis on the heuristic for classifying exceptions and the mechanism for efficiently re-initializing static state.* These two mechanisms are important because they have no analogue in any other automatic testing system of which we are aware.

2. Overview and Assumptions

JCrasher takes as input a program in Java bytecode form and produces a series of test cases for the popular JUnit unit test framework. Random testing can be seen as a search activity: JCrasher is searching for inputs that will cause the target program to crash. The search is exhaustive under user-specified or inferred constraints, like a maximum depth of method chaining. This search space for a specific method is the method's *parameter-space*: the different points represent values for the method's parameters.

The representation and traversal of the parameter-space in JCrasher is type-based. To test a method, JCrasher examines the types of the method's parameters and computes the possible ways to produce values of these types. JCrasher keeps an abstract representation of the parameter-space in the form of a mapping from types to either pre-set values of the type or methods that return a value of the type. This mapping is the data structure that lets JCrasher estimate how many different tests it can produce for a certain method. This information is used for planning: when JCrasher runs with time constraints, it will try to estimate the maximum depth of method chaining so that all tests can run within the time allotted for testing.

JCrasher automates robustness testing and builds on the assumption that robustness can be specified in a more-or-less problem-independent way: *A class should not crash with an unexpected runtime exception, regardless of the parameters provided.* JCrasher automatically provides a huge number of different parameters. We have coded our intuitive specification of "unexpected runtime exception" as a heuristic into JCrasher. The results of each test are filtered by the JCrasher runtime according to a robustness heuristic that determines whether an exception likely constitutes an error or not. Such heuristics are necessary because not all undeclared exceptions correspond to errors in the program: the error may well be in the usage

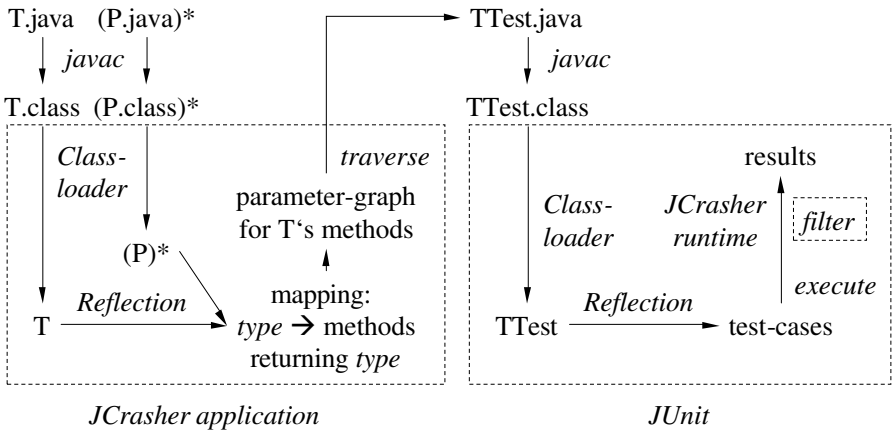


Figure 1. How a class `T` can be checked for robustness with JCrasher. First, the JCrasher application generates a range of test cases for `T` and writes them to `TTest.java`. Second, the test cases can be executed with JUnit, and third, the JCrasher runtime filters exceptions according to the robustness heuristic. * denotes the Kleene star operator.

of the code under test—a precondition violation. In the absence of explicit preconditions it is not generally clear whether an exception is a sign of programming error—the best we can do is offer reasonable heuristics.

Much of the value of the JCrasher approach stems from the fact that public methods are an interface to unknown code. This interface can be manipulated in complex ways and it is the programmer's responsibility to enforce the preconditions of a public method and report the error. Even if a public method does not fail under use within a specific program, it may export an interface that is inconsistent and can fail under other reasonable inputs—boundary cases, for example. JCrasher treats each class of an application as a library class and tries to detect errors both for existing uses of a class within an application and for other possible uses. Of course, this approach is conservative and probably overkill for common programming patterns that ensure that all callers of a method respect its preconditions. Nevertheless, it is the programmer that decides what code he/she wants to test under the stringent criteria of JCrasher.

We call the JCrasher testing approach "robustness testing" because it is most useful in determining whether the public methods of a class detect errors early or let them go unnoticed for a long time. The JCrasher heuristics are defined so that they detect whether an erroneous initial input is propagated deep in the code before it causes an exception. Although this case may not be a bug in practice, it is certainly a failure in terms of robustness: the assumption is that good code detects erroneous input data as early as possible, preventing processing them—possibly violating a program's invariants—and detecting the error much later.

The process of automated robustness testing with JCrasher is illustrated by Figure 1. The user wants to check a Java class `T.class` for robustness. In a first step he/she invokes the JCrasher application and passes it the name `T` of the class to be tested. The JCrasher class loader reads the `T.class` bytecode from the file system, analyzes it using Java reflection [10], and finds for each of the methods declared by `T` and by their transitive parameter types `P` a range of suitable parameter combinations. It selects some of these combinations and writes these as `TTest.java` back into the file system. After compiling `TTest.java` the test cases can be executed with JUnit. Exceptions thrown during test case execution are caught by the JCrasher runtime, which is part of the JUnit test execution by inheriting from JUnit's `TestCase` class. Only exceptions found to violate the robustness heuristic are passed on to JUnit. JUnit collects these exceptions and reports them as errors to the user.

The above process generalizes to multiple classes straightforwardly. If the user passes a set of classes as input to JCrasher, the analysis will examine combinations of all their methods in order to construct testing inputs. In this way, the class can be tested within its current application environment instead of being tested in isolation.

By producing JUnit test cases, JCrasher enables human inspection of the auto-generated tests. Particularly successful auto-generated tests can then be kept as part of a regression test suite. Additionally, JUnit represents an active community and integrating with it enables mutual benefit. For instance, a number of extensions[†] to JUnit already exist.

3. Test case generation

In this section we describe in detail the test case generation logic of JCrasher. Note that this part of the system has engineering novelty but little conceptual novelty: similar ideas have been used in different settings—scenarios for testing GUIs [6], for example. Nevertheless, this section is necessary for a complete description of the system.

For each method `f` declared by a given class `T`, JCrasher generates a random sample of suitable test cases. Each test case passes a different parameter combination to `f`. This generation is done by the following steps. First, JCrasher uses Java reflection to identify method parameter types, types returned by methods, subtyping relations and visibility constraints—`private`, for example. JCrasher adds each accessible method to an in-memory data-structure mapping a type to some pre-set values and methods returning the type. Second, using the above mapping, JCrasher determines for each method `f` how many and which test cases to generate and writes them as a JUnit test-class `TTest.java` into the file system.

The remainder of this section introduces the JCrasher representation of the parameter-space, followed by the JCrasher test case selection algorithm.

[†]See JUnit's web-site, `http://www.junit.org`

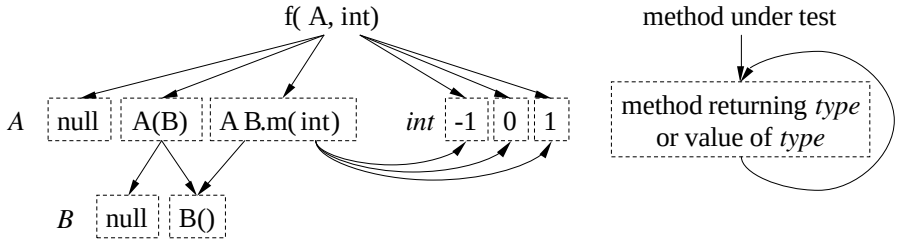


Figure 2. Left: example parameter-graph for method `f(A, int)` under test. Right: key. JCrasher has found `A`-returning methods `A(B)` and `B.m(int)`, and the `B`-returning method `B()`. For `int` JCrasher uses the predefined values `-1`, `0`, and `1`; `null` is predefined for reference types like `A` and `B`. JCrasher derives test cases for `f` from such a parameter-graph by choosing parameter combinations, for example, `f(null, -1)`, `f(null, 0)`, `f(null, 1)`, `f(new A(null), -1)`, ..., `f(new B().m(1), 1)`.

3.1. Parameter-graph

Imagine that JCrasher analyzes a method `f` of a class `C` with signature `f(A1, A2, ..., AN) returns R`. From this, JCrasher infers that in order to test this method it has to know how to construct values of types `C`, `A1`, `A2`, ..., `AN`. Additionally, JCrasher infers that it can construct an object of type `R` or any of `R`'s supertypes—the classes `R` extends and the interfaces it implements, as long as it can construct a `C` and an `A1` and an `A2`, etc.

Additionally, JCrasher has implicit knowledge of how to create certain well-known values of different types. For instance, JCrasher knows that it can use the `null` value for any object type. Similarly, a few pre-set values are used to test primitive types—for example, 1.0, 0.0, and -1.0 for `double`.

One way to encode the above knowledge is as Prolog-like inference rules. Every method and every well-known way to construct values of a type can be seen to correspond to such an inference rule. For instance, a method `f(A1, A2, ..., AN) returns R` in class `C` corresponds to a rule `R` ← `C`, `A1`, `A2`, ..., `AN`. Constructing the well-known value 1.0 of type `double` corresponds to the rule `double` ← `1.0`. Creating test cases corresponds to a search action or Prolog-like inference in the space induced by these inference rules.

Instead of storing such inference rules in text form, we represent them as a graph data structure that we call the *parameter-graph*. This representation is convenient both for computation and for illustration. The parameter-graph is computed by examining the methods of the current program under test. Figure 2 illustrates the concept of test case generation from a parameter-graph. A node `f(A,int)` in this graph means that to test method `f` we need both a value of type `A` *and* a value of type `int`. An edge in the graph goes from a type to a method or a well-known value and reflects an inference rule, or a way to get a value of the type. So there is an edge from type `A` to method `m` if `m` can be used to produce a value of type `A`. Multiple edges with the same source node are alternative ways to create a value of this type.

The parameter-graph is an abstract representation of the parameter-space of the program's methods. Creating different parameter combinations for a method under test can be done by traversing the graph. As discussed in the next section, the representation is useful because it allows us to easily bound the depth of method chaining. The following properties of this approach are worth noting.

- Ideally, each test case should provide a method with a different combination of its parameter-types' value-range. In our approach it is possible that two or more test cases produce the same value. This can happen if a `T`-returning method returns the same value for different parameter combinations, or two `T`-returning methods have overlapping return-value ranges.
- If test cases are picked at random then a method with more parameter combinations is tested more often. This is advantageous, as more parameter combinations tend to produce a bigger variety of parameter state. It can be assumed that the method is more complicated as it needs to handle more cases. It is good to generate more test cases for a more complicated method.
- Possible side-effects of methods via variables are ignored in test case selection. JCrasher does not attempt to deliberately search the space of possible side-effects by exploring all possible combinations of method calls. For this reason void-returning methods are currently excluded from the parameter-graph. This limitation is entirely pragmatic: if void-returning methods are considered, the test parameter space becomes huge very quickly. The issue of side-effects between test cases is discussed separately in Section 5.

3.2. Test case selection

By representing the parameter-space implicitly—via the parameter-graph—we can efficiently compute the number of test cases for a given search depth, that is, compute the size of the parameter-space. Similarly, we can have random access to the test cases. Figure 3 illustrates the size computation of a sub-parameter-space.

Computing the size of the parameter-space without creating all possible tests is important because of the need to configure the testing process for different time allocations. Typically, the user of JCrasher knows how much time is available for testing—five hours, for example. From this, JCrasher can determine how many test cases can be generated. Although this depends on the program under test, we use the rule of thumb of one to two million test cases in five hours, which has been realistic for our testing environment. Knowing the total size of the parameter-space, JCrasher can select to output randomly a certain percentage of the test cases.

4. Test Case Execution and Exception Filtering

Each test case consists of one or more method or constructor invocations contained in a single `try` block:

```
public void test1() throws Throwable {
```

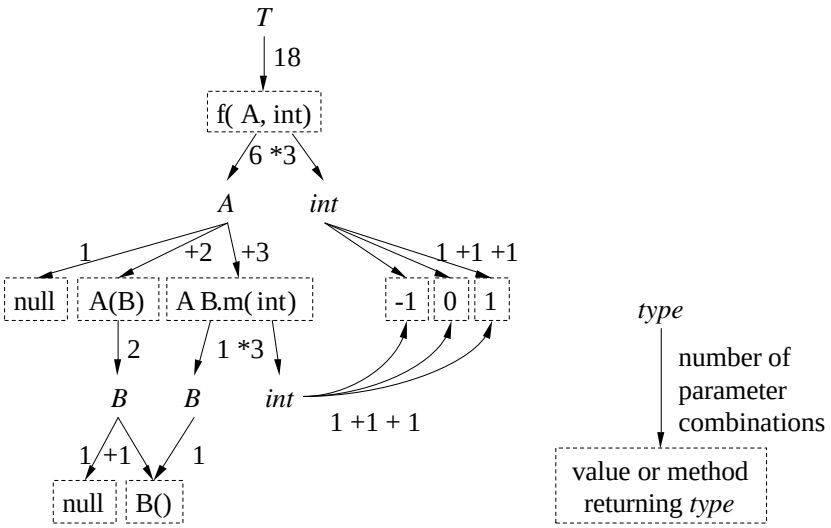


Figure 3. The size of a sub-parameter-space is calculated bottom up by adding the sizes of a type's value and method spaces and multiplying the sizes of a method's parameter type spaces. Left: example continued from Figure 2. Right: key.

```
  try {
    //test case
  }
  catch (Exception e) {
    dispatchException(e);
  }
}
```

Each `Exception` thrown by a test case is caught and passed to the JCrasher runtime. According to the heuristics given in this section the runtime either considers the exception a bug of the code under test and passes the exception on to JUnit or considers it an ill-formed test case—that is, failure to provide legal inputs—and suppresses the exception. The heuristics that the JCrasher runtime uses to make this distinction are part of of the novelty of JCrasher. These heuristics take into account the type of the exception and the method call history that led to the exception. We describe these heuristics next.

4.1. Classification of `Throwable`

Figure 4 illustrates how JCrasher classifies each caught `Throwable`.

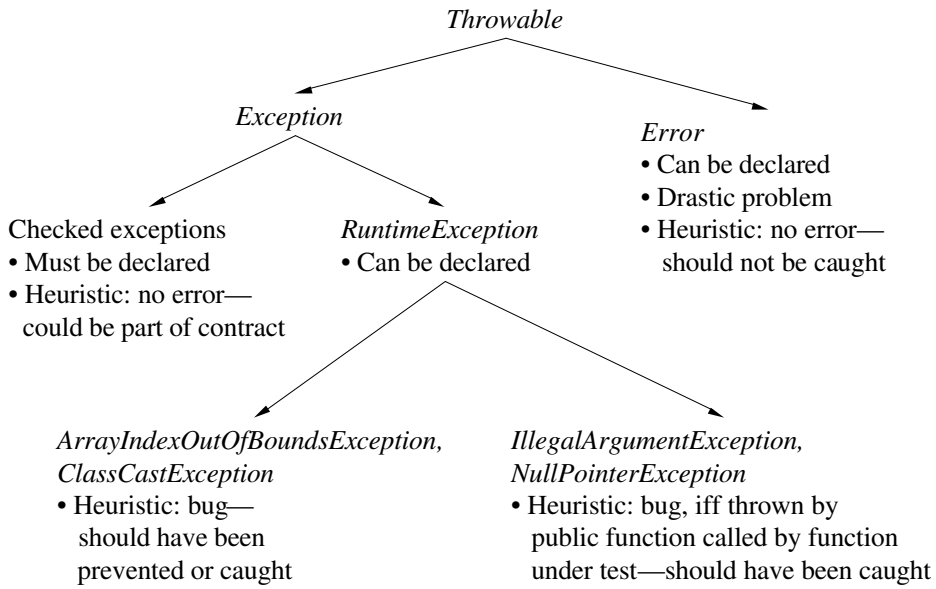


Figure 4. Classification of the Java sub-class hierarchy of `java.lang.Throwable` according to the robustness heuristic coded into JCrasher. An arrow is drawn from super-type to direct or transitive subtype. `Error`, `Exception`, `RuntimeException`, and the depicted subclasses of `RuntimeException` are, like `Throwable`, classes of package `java.lang`.

- Java suggests throwing a `java.lang.Error` in case of a serious problem. For example, an `OutOfMemoryError` is thrown if the JVM runs out of memory. Catching an `Error` is considered a problem orthogonal to the test execution. An `Error` is always passed on to JUnit.
- *Checked exceptions* can only be thrown or passed on by methods or constructors that declare to throw them. But it is not formally stated under which conditions the exception is thrown—that is, whether throwing such an exception should be considered an error or just a non-linear transfer of control. Therefore we cannot automatically decide if a checked exception was to be expected or not. A checked exception is never passed on to JUnit.
- *Unchecked exceptions* are of type `java.lang.RuntimeException`. Each and every method and constructor may throw an unchecked exception, regardless of whether this has been declared in the method's signature or not. There seem to be the following two kinds of unchecked exceptions.
- An exception of the *first group of unchecked exceptions* is typically thrown by low-level functions mostly implemented by the JVM. For these language features no user method is called and therefore no function frame is put onto the execution stack. Therefore the

user method providing the wrong precondition is on top of the execution stack. JCrasher considers this exception a bug of the method `f` under test. If the exception is thrown by a method called by `f` then `f` is to blame for not catching it—and handling it. Such exceptions are `ArrayIndexOutOfBoundsException`, `NegativeArraySizeException`, `ArrayStoreException`, `ClassCastException`, and `ArithmeticException`; all of them are of package `java.lang`.

- An exception from the *second group of unchecked exceptions* usually indicates that the precondition of the called method or constructor has been violated. Such exceptions include `IllegalArgumentException` and `IllegalStateException` from package `java.lang`. Here the method whose preconditions have been violated is the top stack element. This exception is analyzed further to determine which method has thrown it.
  - If the exception is thrown directly by the method under test, then it is reasonable to assume that our random input is not valid for this method [11]. For instance, throwing an `IllegalArgumentException` exception is the recommended Java practice for reporting illegal inputs.
  - If the exception is thrown by a non-public method that is called—perhaps transitively through other non-public methods—by the method under test, JCrasher does not consider it an error: non-public methods can be viewed as an implementation detail, akin to executing code within the method under test. Clearly, this distinction is somewhat arbitrary: causing a non-public method to throw an unchecked exception may well indicate a robustness failure—the code's invariants may have been violated before the exception is thrown. Nevertheless, in experimenting with JCrasher we found that considering such exceptions to be errors precludes many common and useful programming patterns. For instance, non-public methods are often called in order to enforce a public method's precondition—by throwing an exception if the input does not satisfy it.
  - If the exception is thrown by a public method transitively called by the method under test, then it is considered a bug—the method under test is to blame for calling other public code that throws exceptions that the method under test does not expect. Implementing robust code means that a precondition error for one access point—that is, a public method—does not propagate to precondition violations for other access points.

A few special cases are worth mentioning. Just as non-public methods are considered "owned" by their most immediate public caller, superclass constructors are considered "owned" by their subclass constructor. If a constructor defined in a superclass throws an exception, this is equivalent to the subclass constructor directly throwing the same exception. The reason is that superclass constructors are called before subclass constructors in Java, thus the author of the subclass code cannot ensure that no exception will be thrown on illegal data.

One more special case concerns `java.lang.NullPointerException`. After experimenting with JCrasher we have decided to move `NullPointerException` from the first group of runtime exceptions to the second one. A `NullPointerException` is typically thrown by the JVM but often stems from passing `null` as a parameter to a method or constructor. So throwing,

which includes passing on, a `NullPointerException` is viewed as similar to throwing an `IllegalArgumentException`.

Finally, we should note that the problem of assigning blame for errors in the absence of precondition specifications, or when the failure condition is only caught by a precondition of a nested call, is a standard one in testing—in the Java context it was described independently in Reference [11], for example. The newly added Java assertion facility does not address the problem as it does not specify whether the assertion failure is the responsibility of the caller—due to illegal input—or the responsibility of the callee.

5. Resetting Static State

A large part of the design and implementation effort of JCrasher has gone into making it an efficient and scalable tool. Most of the JCrasher tests are very short in practice—they consist of a method call with arguments that are returned by other method calls, etc., but typically only up to a depth of three to five. More complex tests are not too meaningful due to the enormous size of the test space for all but the simplest classes. Since JCrasher tests small units of functionality for relatively "shallow" errors, it makes sense to perform all its tests in a single instance of the Java Virtual Machine, thus avoiding the expensive tasks of process creation, JVM loading, etc. Nevertheless, this raises the problem of dependencies among test cases. Without special care, a test case will end up setting the static state on which subsequent test cases will execute. This interference is undesirable: ideally, we would like to know whether a method fails in a well-defined initial state. Otherwise it would be hard to detect the real cause for the violation of a program's invariants.

To enable different test cases to execute without interference in a single virtual machine instance, we experimented with two main ideas: using a different class loader for each test and 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. Both techniques aim at re-initializing all static data in the classes under test. That is, our techniques reset the in-memory, user-level state introduced by previous tests. A limitation is that external state—files or data stored in a database, for example—or static state in system classes cannot be reset. Nevertheless, methods that are sensitive to external or system state are typically too complex to benefit by a random testing approach.

The problem of resetting static state is interesting for applications very different from testing. Although, as we argue later, the case of JCrasher is unusual in that it does not require full correctness: a minor relaxation of correctness only risks introducing some false positives and may be desirable if it yields much better performance. For instance, Reference [12] proposes a JVM design that allows fast re-initialization without process creation overhead for server applications. In our setting, working with an unmodified, general purpose JVM is a huge advantage for the ease of deployment of JCrasher. To our knowledge, the comparative merits of the approaches described here have not been discussed before. Resetting static state for testing scalability is one of the novelties of JCrasher.

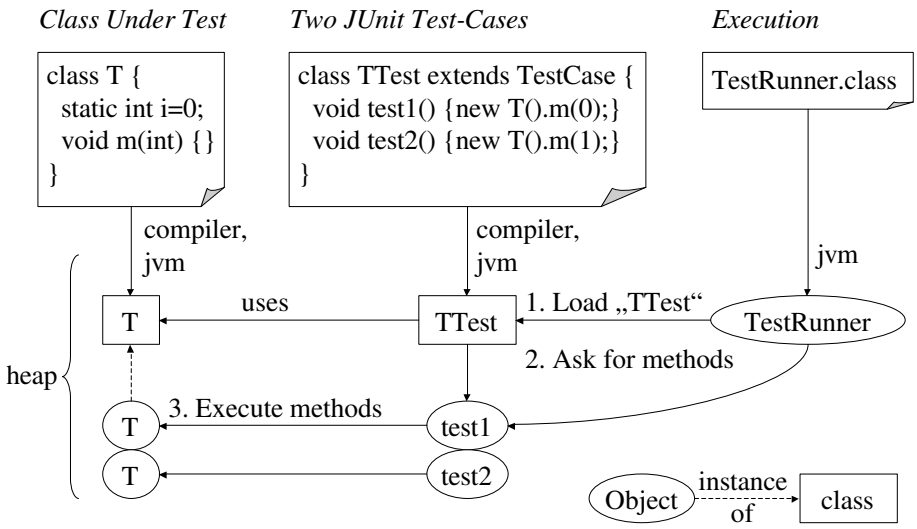


Figure 5. How JUnit executes two test cases `test1`, `test2` of a JUnit test-class `TTest`, testing class `T`. JUnit loads `TTest` by name via a class loader and retrieves and executes its test cases via reflection. Class `T` is loaded on test case execution as instances of `T` are created.

5.1. JUnit

We first describe the JUnit machinery very briefly. Both of our approaches to resetting static state require minor modifications to JUnit. This is somewhat undesirable since it means that a patched version of JUnit is needed during JCrasher testing. Nevertheless, the changes are very small—a few lines patched—and the main benefits of using JUnit as the JCrasher back-end are preserved: generated test cases remain in JUnit format and if they prove to be interesting for regression testing they can be used with an unmodified version of JUnit.

JUnit automates the execution of test cases and facilitates the collection of test case results, but does not offer any assistance in implementing test cases. The JUnit user provides a test class—a subclass of `junit.framework.TestCase`—with test methods that exercise the functionality of different classes. Figure 5 illustrates how JUnit works. In this example, JUnit represents a test case through a method `test1` of a test-class `TTest`. The predefined JUnit code uses reflection to find methods whose name begins with "test" and executes them. JCrasher test cases correspond to individual "test" methods as described in Section 4.

5.2. Multiple Class Loaders

The first option offered by JCrasher for resetting static state consists of loading each test case with a different class loader. Using multiple class loaders is a Java technique for integrating

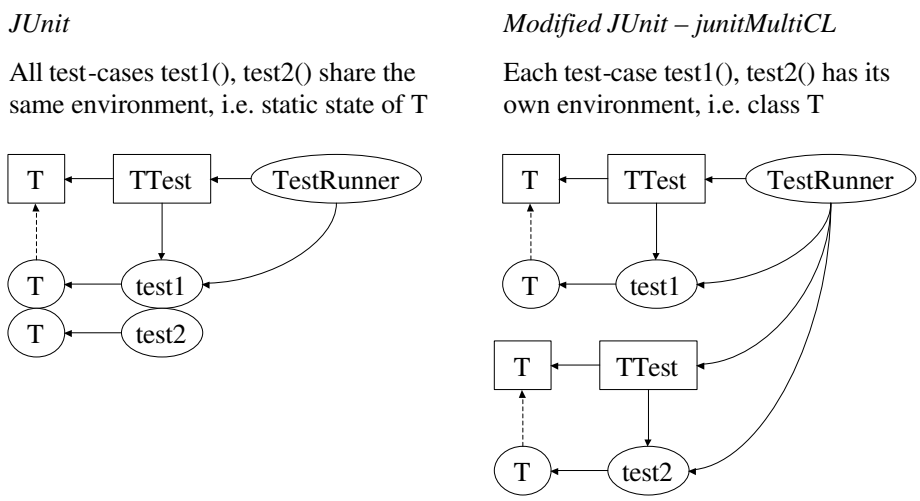


Figure 6. Concept of how to modify JUnit to undo changes of class state

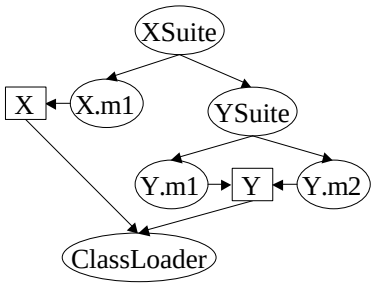
the same code multiple times, with different copies of its static state. In essence, the same class bytecode loaded twice with different class loaders results in two entirely different classes inside a Java VM.

Figure 6 illustrates the concept. It would seem that the modifications to JUnit for integrating the multiple class loaders idea are very minimal and localized. Indeed, our first JUnit patch with early versions of JCrasher consisted of about 10 lines of source code. Nevertheless, we quickly found out that this version was not scalable: due to the JUnit representation of test cases and their interdependencies, older classes never became unreferenced and thus could not be garbage collected. This limited the scalability of JCrasher to a few hundreds of test cases, well below our goal of many hundreds of thousands of test cases per JVM process. Of course, multiple JVM instances could be used in sequence, in order to cover all test cases, but this would require more complex coordination outside the JUnit framework—an executable program or script that starts virtual machines, for example—which is an undesirable solution for deployment/portability purposes.

Our final solution changes the JUnit execution model more substantially, as shown in Figure 7. Instead of having a single "Test Suite" object that recursively points to all the tests to be executed, our modified version of JUnit has independent suites, each supporting a small fixed number of test cases—we have been using a single test case per suite. Once a suite gets executed, all memory it occupies—together with loaded classes—can be garbage collected. The only exception is for test cases that actually resulted in errors. In that case, JUnit stores a reference to the offending test case in its error report and the garbage collector cannot reclaim that memory.

JUnit

1. One classloader loads all test-cases and all used classes
2. One suite-hierarchy is executed



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

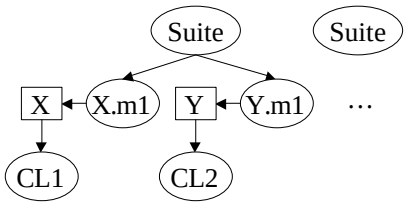


Figure 7. How JUnit's execution model is modified to undo changes to class state

The multiple class loader approach is a reasonable compromise between conciseness and efficiency. Nevertheless, it is still rather inefficient since all user classes need to be re-loaded every time just to execute a new test method on a clean slate.

5.3. Load-Time Re-Initialization of Java Classes

A second alternative in JCrasher for resetting static state is to imitate the JVM's class initialization algorithm in user space. This will allow re-initializing the same set of classes, existing from previous test cases. This gives much better performance as it saves the time needed for loading and unloading classes. Furthermore, it yields better scalability in terms of memory requirements in the case of test cases resulting in errors. Recall that with multiple loaders, multiple copies of the same classes have to be kept for executions that did cause an error.

A significant insight for the rest of this section is that our solution does not need to be fully correct with respect to the semantics of Java VM initialization, as long as it is correct for the vast majority of test cases. Testing with JCrasher is an optimistic, "nothing-to-lose" approach. The system picks some test inputs randomly and attempts to discover robustness errors. As long as the vast majority of the test cases execute with correct Java semantics, it should not matter if a few false positives—failed tests that do not correspond to errors—are introduced. False negatives matter even less as they are inherent in the random input approach anyway. Thus, it is often preferable to have a very fast but not fully accurate solution. Indeed, the current JCrasher implementation does not perform fully correct static re-initialization

although it works correctly for the vast majority of programs. In the next section we will discuss how its correctness can be improved.

The key idea is to implement a load-time class initialization algorithm—similar to the one performed by the JVM [13, Section 2.17.5]—in the JCrasher runtime. Executing this procedure before each test case execution re-initializes the static state of all previously loaded classes each time to the same value. The required elements of such a load-time initialization algorithm are the following. First, a list of the classes in the order in which they should be initialized. Second, the ability to reset the values of the static fields of each of these classes to the default before each test case execution. All zero bits represents the default value: `null`, zero, or `false`. Finally, the ability to execute the variable initializer of each static field before each test case execution. The variable initializer of a static field is compiled into the `<clinit>()` method of the class declaring the field. The `<clinit>()` method cannot be called by Java code as its name is not a valid method name.

In order to implement class re-initialization, JCrasher again makes some modifications to JUnit. The JUnit class loader is replaced by a special class loader that performs modifications of the class bytecode before loading it—we use BCEL [14] for bytecode engineering. Classes belonging to the Java platform, JUnit, or JCrasher are excluded from the following treatment by the class loader.

- If the class has a `<clinit>()` method, this is copied to methods `_clreinit()` and `_clinit()`. The former method is the re-initializer, while the latter will be used as the original initializer. If no static initializer exists, empty `_clreinit()` and `_clinit()` methods are added to the class.
- The `_clreinit()` method is modified to differ from the original static initializer to avoid attempting to reset final fields.
- A static initializer, `<clinit>()`, method is added to the class. This static initializer calls the original static initialization code, `_clinit()`. On return from that code, the static initializer registers the fact that static initialization ended for this class in a JCrasher-maintained data structure. The ending order of initializations will be the same as the order of re-initializations before future tests.

The re-initialization occurs after the end of each test case execution. JUnit calls via `TestCase.setup()` a JCrasher-runtime function. This function first sets the static fields of the registered classes to all zero bits using Java reflection. Then the `_clreinit()` method of each class is executed. The classes are re-initialized in the order their original initializations finished in previous tests. This is not correct Java initialization, for two reasons: first because of the possibility of cyclic dependencies between classes and second because this re-initialization is eager—it happens up front.

In the case that static data among classes have cyclic dependencies, the order of re-initialization could be incorrect. In some cases, the loading order of classes could affect the result of static initialization. Thus different test cases could need to be initialized with different static data although they use exactly the same classes—but the code is such that they are loaded in different order. For a standard cyclic dependency example [15], consider the following classes:

```
class A {static int a = B.b + 1;}
```

```
class B {static int b = A.a + 1;}
```

In a regular execution, if the initialization of A is started before the initialization of B, then B is initialized before A and a=2, b=1; else a=1, b=2. So if B has been loaded before A and the JCrasher-runtime would run the B-initializer, then the A-initializer: a=1, b=2. Code with cyclic static dependencies is *extremely rare*—the study of Kozen and Stillerman [15] examined a few libraries and found no examples, and such code should be avoided anyway.

The eager character of the JCrasher re-initialization represents a more serious departure from the Java initialization semantics. The Java semantics enforces the lazy loading of classes. Static data get initialized as classes get loaded. Furthermore, static data can be initialized to values dependent on dynamic data computed by the code of previously loaded classes. Thus, eager re-initialization—before any other computation has taken place—may yield different values. Nevertheless, most Java programs do not rely on the timing of class loading, so we expect such errors to be rare.

As discussed earlier, the lack of full support for the Java semantics is not a big problem in practice for an optimistic testing approach like that of JCrasher. The performance advantages of the re-initialization approach far outweigh the small danger of false positives in the testing process. Furthermore, in practice, it is unlikely that such false positives will be introduced during regular development.

5.4. Performance

We have conducted preliminary experiments on the performance of the different static state resetting techniques. Roughly speaking, the multiple class loading approach is more than twice as fast as re-starting the entire Java VM if only a couple of classes are being reloaded. In contrast, the less safe re-initialization approach of Section 5.3 is over 20 times faster than multiple loading, making the resetting time be negligible with respect to other overheads.

Of course, whether the resetting overhead matters depends on the time that each test takes. Most tests executed when testing an actual application are very brief—each lasts on average a few milliseconds with some being significantly shorter and a few happening to "go deep" in application functionality and take much longer. Of course, the user can select to supply the top-level classes of an application to JCrasher, in which case even a single test may take arbitrarily long to complete.

Additionally, the current implementation of JCrasher is relatively heavyweight. We output all test cases as files on the disk, JUnit has to load each test file and execute it, etc. For efficiency, JCrasher produces test files that contain 500 test cases each. One can easily imagine a more specialized, tuned to batch execution runtime, where test cases will be executed with less overhead. Nevertheless, *even in the current, heavyweight JCrasher execution model, the relative overhead of re-initializing classes can be significant.* This relative overhead would only be more significant with a faster runtime.

More specifically, in our testing environment—a 1.2 GHz Intel mobile Pentium 3 processor with 512 MB RAM running Windows XP and a 12 ms avg., 100 MB/s hard disk—the average time taken to execute a test with JUnit is a little below 5 ms. Starting a Java VM that will execute a trivial class takes 170 ms. Starting a VM and executing a trivial JUnit test takes

270 ms—that is, loading the JUnit classes, the test class and one application class under test and running the JUnit testing code. With the multiple loading approach, using a new loader to re-load a single application class takes 120 ms. That is, the multiple class loaders approach saves the 270 ms of JVM startup time and JUnit re-loading time. Nevertheless, even going to disk and reloading a single class file introduces enough overhead to reduce the overall benefit to about 150 ms. Still, the multiple class loading approach is more than twice as fast as restarting the JVM.

The re-initialization approach through calling the static initialization code, discussed in Section 5.3, is significantly faster. The time taken by the JCrasher machinery for re-initializing a class with 10 static fields is 0.06 ms, which is negligible. Although this number depends on the complexity of static initializers, the approach is clearly one of minimal overhead and the time savings are dramatic with respect to the 5 ms it takes JUnit to execute a simple test. Overall, the re-initialization approach enabled the latest version of JCrasher to execute tests an order of magnitude faster than previous versions.

5.5. Alternatives and Future Directions

There are several directions along which the current JCrasher re-initialization algorithm can evolve. These ideas can offer benefits such as guaranteed correctness or the ability to use an unmodified version of JUnit. Pursuing these directions is part of future work.

First, our load time re-initialization approach for resetting static state is code-driven: we re-run the initialization code for each class. Instead, one can imagine a data-driven approach were all static state is saved and later restored. The two approaches would have the standard trade-offs of memoizing values vs. recomputing them. Nevertheless, because of lazy class loading, a data-driven approach will require sophistication, in order to support updating the snapshot when previously unknown classes are encountered.

Second, an interesting approach, which we plan to explore in future work, would be to revert to a correct but more expensive technique—like using multiple class loaders—for re-initializing static state but only for tests that fail with the fast but not fully correct load-time reinitialization technique.

Third, JCrasher could use a static analysis algorithm to determine the static dependencies between classes. One such algorithm is described by Kozen and Stillerman [15]. Although using the output of the algorithm to infer the order of re-initialization will not fully respect the Java semantics, Kozen and Stillerman argue that the cases that will differ constitute "bad" circularities that are probably programming errors.

Fourth, the rewrite of classes so that they are re-initializable could occur off-line—in a separate compile phase—instead of at load time. This has the disadvantage of assuming that all classes that ever need to be loaded for a test are known to JCrasher and available beforehand—dynamic loading is not supported. Nevertheless, the advantage is that the test cases then become regular JUnit test cases—they can be executed with an unmodified version of JUnit.

Fifth, JCrasher could get correct execution semantics by simulating the JVM class initialization procedure exactly. This would require extensive bytecode modification, which will introduce significant overhead. For instance, the event that will trigger the initialization of a class may be the creation of an instance, meaning that all object creations throughout

the program under test need to be rewritten to first check whether the instantiated class has been re-initialized. We believe that this approach would be overkill in terms of implementation complexity and overhead, for very little practical benefit.

6. JCrasher in Practice

In this section we discuss practical considerations concerning the use of JCrasher, as well as some experiments and results.

6.1. Pragmatics

We can loosely identify two major modes of using JCrasher: the *interactive* mode and the *batch* mode. There is no real technical difference between the two modes: interactive mode is the same as batch running with parameters such that the test space is small. JCrasher is mainly used interactively as a plug-in to the Eclipse IDE, so that a few mouse point-and-click operations can generate simple test files for a class—to call an `int` routine with some random integer inputs, for example. This is a useful feature because newly developed code is likely to have "shallow" bugs that some random testing will uncover. JCrasher provides a way to test the new code under many inputs without running the entire application, effectively automating some of the easiest cases of unit testing. Furthermore, JCrasher requires practically no learning investment, thus serving as a good introduction to testing and to JUnit.

In batch mode, JCrasher is given a set of classes to test and a maximum nesting depth for the method calls to be attempted. As described earlier, JCrasher will construct the parameter-space abstractly and will then pick tests to execute within the time allotted. It is worth noting some of the scaling issues. The re-initialization approach for static data enables much faster execution of test cases than previous approaches. For a five hour testing period, one million tests is typical. Again, though, this number depends heavily on the specific application classes chosen for testing, as the net application time per test could range from a few nanoseconds to arbitrarily long. Hence, JCrasher can easily produce large amounts of file system data. For 1 million tests the disk space occupied is 200-300 MB for source code and another 100 to 150 MB for bytecode. Also, compiling the test cases is expensive—about 7 ms per test case on average in our environment. One could argue that the JCrasher batch mode could have higher performance if it avoided the explicit use of JUnit and did not store the test cases as files. If, instead, the testing was done by JCrasher while the test cases are being produced and only the failed test cases were exported as JUnit files, then performance could increase significantly. Although for inspection reasons it is beneficial to reify all test cases as files, we plan to explore the above alternative in future work. Note that a faster runtime would make our fast static state resetting techniques even more important.

JCrasher is also useful, in both the incremental and batch mode, in learning to use third-party libraries. Using JCrasher on unknown code gives a low-effort way to explore the behavior of the code, for example, with respect to null values. Furthermore, just by skimming the generated test cases one can get a good idea of the parameter-space and the way methods are

combined in the unknown API. In general, in some ways JCrasher can be viewed as more of an empirical analysis tool than a testing tool.

An important issue is the grouping of exceptions. Since JCrasher can produce many isomorphic test cases, thousands of exceptions could be caused by the same error—or non-error. We currently have a heuristic way to group exceptions before presenting them to the user. The grouping is done according to the contents of the call-stack when the exception is thrown. Grouping is significant from the usability standpoint because it eliminates many of the complexities of dealing with false positives during the automatic testing process. For a hypothetical scenario, imagine that 1 million tests are run and 50,000 of them fail. It is likely that the independent causes of failure are no more than a handful of value combinations. By grouping all exceptions based on where they were thrown from and what other methods are on the stack, we offer the user a scalable way of browsing the results of JCrasher and separating errors from false positives.

6.2. Use of JCrasher

This section presents experiments on the use of JCrasher. We demonstrate the types of problems that JCrasher finds, the problems it does not find, its false positives, as well as performance metrics on the test cases.

In our experiments, we tried JCrasher 0.27 on the Raytracer application from the SPEC JVM benchmark suite, on homework submissions for an undergraduate programming class, and on the uniqueBoundedStack class used previously in the testing literature [16, 17]. Xie and Notkin refer to this class as UB-Stack, a name that we adopt for brevity. We have changed all methods of UB-Stack from package-visible to public as JCrasher only tests public methods. Besides this, we did not modify the testees for our experiments. JCrasher 0.27 and the selected test cases are available on the JCrasher project web site.

As described previously, JCrasher attempts to detect robustness failures, but it is not fully accurate and robustness failures do not always imply bugs. Hence, the user needs to inspect the JCrasher output and determine whether the test input was valid, whether the program behavior constitutes a robustness failure, and whether the program behavior constitutes a bug. In general, JCrasher has few false positives in its effort to track robustness failures: the heuristics of Section 4 work quite well. We present next selected instances of robustness failures, although not all are bugs. In the case of student homeworks, we had access to many more submissions but we limited the number of testees to a small set that covers all the different exception types reported, but did not otherwise bias the selection, which should be fairly representative. By picking only a few testees we could easily examine the JCrasher reports and determine whether they constitute bugs. In general, JCrasher detected a few shallow problems in the testees. These problems can usually also be found during unstructured testing or by running simple test cases. The value of JCrasher is that it automatically detects these shallow problems and therefore allows the developer to concentrate on interesting problems.

Tables I and II give program and testing performance metrics for the testees. We have conducted our experiments in the testing environment described in Section 5.4. The numbers shown are for the JUnit text interface—which gives faster test execution than the graphical interface. We show how many tests JCrasher generates, how long it takes to generate these

Table I. Results of using JCrasher on real programs (testees). *Test cases* gives the total number of test cases generated for a testee when searching up to a method-chaining depth of three. *Crashes* denotes the number of errors or exceptions thrown when executing these test cases. *Problem reports* denotes the number of distinct groups of robustness failures reported to the user—all crashes in a group have an identical call-stack trace. *Redundant reports* gives the number of problem reports we have manually classified as redundant. *Bugs* denotes the number of problem reports that reveal a violation of the testee's specification.

Testee			Tests				
Class name	Author	Public methods	Test Cases	Crashes	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 II. The execution time and disk space required for generated test cases. *Report* is the output of the JUnit text interface redirected to a text file.

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

tests, how much disk space they occupy, how long it takes to execute them, how many problems are reported to the user, how much disk space these reports occupy, how many of these reports we consider redundant, and how many reports can reasonably be considered bugs.

To enhance readability we have condensed the layout of the source code shown in this section. Omitted code is represented by `//[..]`.

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. Since we have no specification for the application, we cannot tell what behavior constitutes a bug, but we try to make reasonable estimates.

In our testing of the Canvas class of the Raytracer, we found a constructor method that is particularly interesting. The constructor below throws a `NegativeArraySizeException` when passed parameters `(-1, 1)`.

```
/* spec.benchmarks._205_raytrace */
public Canvas(int width, int height) {
  Width = width; Height = height;
  if (Width < 0 || Height < 0) {
    System.err.print("Invalid window size!" + "\n");
    //System.exit(-1);
  }
  Pixels = new int[Width * Height];
}
```

In this example, it is clear that passing in a negative `width` would be an erroneous input. Indeed, the original developer of this code agreed that it would be an error to continue after this input—a line to exit the program existed. Nevertheless, the exit command is now commented out, possibly because the Raytracer is used as part of a larger program. But it is clear that continuing with a negative width or height is a robustness error. It is not even necessarily the case that the error will be caught during the allocation of the `Pixels` array: if both `height` and `width` are negative, their product will be positive and the allocation will succeed, letting the error propagate further in the program.

The test case below generated by JCrasher calls the following `Write` method of Canvas, which in turn calls its private `SetRed` method. `SetRed` throws 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);
}
```

This is a case of a robustness failure that probably does not represent a bug—it can be argued that the input violates the preconditions of the routine. Nevertheless, it would be a good programming practice to use the new Java assertion facility to enforce the preconditions of method `Write`.

Student Homeworks. We applied JCrasher in testing homework submissions from a second-semester Java class. This is a good application domain, as beginning programmers are likely to introduce shallow errors in their programs. Furthermore, since the tasks required are small, self-contained, and strictly specified—the homework handout describes in detail the input assumptions—it is easy to distinguish bugs from normal behavior. On the other hand, these student programs are usually too small to exhibit interesting constructor nesting depth or a large number of test cases for a single program.

We next describe a few selected cases of robustness failures. One task in homework assignment P1 required the coding of a pattern-matching routine. Given a pattern and a list of integers, both represented as arrays, the routine should attempt to find the pattern in the list. Passing `([0], [0])` to the following method `findPattern` by programmer s1 causes an `ArrayIndexOutOfBoundsException`, although the input is perfectly valid. The cause is a badly coded routine:

```
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]);
      }
}
```

A similar error is reported in a different statement when the input is `([0], [])`, which is also a bug, but can be argued to violate the programmer's understood pre-condition.

Another robustness failure that can be considered a bug is a `NumberFormatException` for programmer s3426's testee P1 when passing String `" "` to the following main method.

```
public static void main(String[] argv) {
```

```
  int tests = Integer.parseInt(argv[0]); //[..]
}
```

This is a common case of receiving unstructured input from the user, without checking whether it satisfies the programmer's assumptions.

Another bug consists of a `NegativeArraySizeException` reported for programmer s8007's testee P1 when passing -1 to the following method `getSquaresArray`.

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

This is a programmer error since the homework specification explicitly states "If the value passed into the method is negative, you should return an empty array."

An interesting case of error is exhibited in programmer s2251's BSTree homework assignment. The BSTNode class contains code as follows:

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

That is, the constructor checks at run-time that the data it receives support the `Comparable` interface. This caused an exception for a JCrasher-generated test case. Changing the signature of the above `BSTNode` constructor from `BSTNode(Object dat)` to `BSTNode(Comparable dat)` fixes the problem. As a result of this change, JCrasher produces only 332 test cases (instead of 2000 before), detects 60 crashes (941), and reports three problems (four). The reason is that JCrasher finds fewer possibilities to create a `Comparable` than an `Object`. The remaining three reported problems for the BSTNode program are caused by passing a `null` reference to a method that does not expect it. Hence, two of the problem reports are redundant and none of the three are bugs.

The rest of our tests from Tables I and II reported similar robustness failures—due to negative number inputs, null pointer values, etc.—that did not, however, constitute bugs.

UB-Stack. We finally tested JCrasher with the UB-Stack class, previously used as a case study in the testing literature. JCrasher does not report any problems for UB-Stack. Stotts, Lindsey, and Antley have hand-coded two sets of eight and 16 test cases, respectively. The smaller set does not reveal any bugs but the bigger set reveals a bug [16]. The bug occurs when executing a sequence of the testee's methods. JCrasher cannot detect this bug since it currently does not produce test cases that explore executing different methods in sequence, just for their side-effects. Xie and Notkin [17] have presented an iterative invariant and test case generation technique, and used the two hand-coded sets as initial sets. For both initial sets they generate test cases that reveal at least one bug in UB-Stack. In the future, it will be interesting to explore the use of JCrasher together with automatic invariant detection techniques.

7. Related Work

Testing has been a topic of study for several decades. Thus, there is an enormous amount of work on testing techniques. Here we will refer selectively to work that either is most similar to JCrasher or is recent and has good further bibliographical references. There are large parts of the testing literature that we will not cover at all—for example, techniques that use compiler-based analyses for computing test coverage, mutation testing, etc.

7.1. JUnit Test Class Generators

Like JCrasher, Enhanced JUnit [8] automatically generates JUnit test classes from bytecode. Like JCrasher, Enhanced JUnit chains constructors up to a user defined maximal depth. 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. For example, when testing class

```
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.

The commercial tool Jtest [7] 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. The maximal supported depth of chaining seems to be three, though. Since there is little technical documentation, it is not clear to us how Jtest deals with issues of representing and managing the parameter-space, classifying exceptions as errors or invalid tests, etc. Jtest does, however, seem to have a sophisticated test planning approach, employing static analysis in order to identify what kinds of test inputs are likely to cause problems. In general, 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.

The JUnit Test Generator [18] 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.

7.2. Providing Random Input

The Ballista project at Carnegie Mellon University has led to automatically crashing different software systems like operating systems [5] or CORBA object request brokers [19]. Ballista

focuses on assessing how effective exceptional input is handled by the software modules under test. For example, Kropp et al. [5] do automatic reliability testing by dynamically providing parameters for POSIX operating system functions. Note that the problems faced at that level are entirely different from those of JCrasher. For instance, the set of supported parameter types is fixed and known in advance. Therefore no discovery or planning is needed. Also, crashing is always an error at the operating system level.

The "Fuzz Testing of Application Reliability" project at the University of Wisconsin at Madison has done black box testing using random input. For example, Windows GUI applications are provided random streams of keystrokes, mouse events and Win32 messages [3]. Win32 messages are normally produced by the Windows NT kernel, for example, one for each mouse click. The test program generates these messages with arbitrarily set parameters and feeds them into the software under test. As with Ballista this approach does not need any type discovery or planning and crashing is always an error.

Claessen and Hughes test pure functions in the Haskell programming language using random input [2]. This is easier than testing arbitrary procedures or methods, as functions by definition cannot have any side effects. There is no state that can change due to the execution of a function.

7.3. Formal Specifications

Much research work on automated software testing has concentrated on checking the program against some formal specification [20, 21, 22]. Of course, the main caveat to this automation is that the formal specification needs to be available in the first place. Writing a formal specification that is sufficiently descriptive to capture interesting properties for a large piece of software is hard. Furthermore, if the specification is reasonably expressive—in full first-order logic, for example, where the quantified variables can range over dynamic values—then conformance to the specification is not automatically checkable. That is, conformance can be checked for the values currently available but checking it for all dynamic values is equivalent to program verification. Therefore, generation of test data is again necessary. A good example is the recent JML+JUnit work of Cheon and Leavens on using the Java Modeling Language (JML) to construct JUnit test cases [11]. Skeletal JUnit test case code that calls the program's methods is generated. The test inputs are user-supplied, however. The integration of JML in this approach is as a rich assertion language—the JML assertion checker is used to determine whether some invariant was violated.

A recent piece of work on generating test inputs automatically from specifications in the Java setting is the Korat system [23]. Korat integrates with the JML+JUnit approach but automates the step of supplying test inputs. The interesting aspect of the system is that only non-isomorphic test cases are generated and that the parameter-space can be pruned if certain parts of the input are irrelevant to the outcome of the test. This aspect is orthogonal to JCrasher and would be very interesting to integrate.

Overall, compared to the above approaches, JCrasher emphasizes a much lower maintenance approach to testing. Instead of expecting formal specifications for deciding what is well-formed input, we rely entirely on type system information. Instead of checking against user-defined tests for errors, we try to heuristically determine when a "crash" constitutes an error. Finally,

we emphasize technical aspects, like the scalability of the testing approach with resetting static state efficiently.

8. Conclusions

We presented JCrasher, a random testing tool for Java programs. JCrasher can be used to discover arbitrary program bugs and it is ideal for robustness testing of public interfaces—that is, making sure that the public methods of a program do not let erroneous inputs propagate far in the implementation.

JCrasher can be used either in batch mode, for random testing of large parts of application functionality, or interactively, as a plug-in to the Eclipse IDE. The Eclipse user can create simple, throw-away test cases with a couple of mouse clicks. The result is that JCrasher can be used interactively to test newly written code. In batch mode, JCrasher executes a huge number of tests looking to "crash" the application with an unexpected runtime exception. Both modes are useful for understanding the behavior of unknown code.

We believe that JCrasher offers some interesting ideas and machinery for application testing. These include the heuristics for determining whether an exception constitutes an error and the mechanisms for resetting static state after the execution of a test case. In terms of testing philosophy, we believe that JCrasher is a valuable tool. It requires very little learning and automates the testing process without any need for program behavior specifications—other than type system info. It represents a view where test cases are plentiful, throw-away resources and only good test cases become selected as eventually valuable assets.

Of course, the longer term research question is a usability one: *will third-party users find JCrasher a useful tool to integrate in their projects?* This is yet too early to tell, but we offer JCrasher as a free tool [24], we intend to keep evolving it from an engineering standpoint, and we hope that different developers will try it in various settings. We have already made efforts to make the tool immediately usable in real world development—the JCrasher integration with Eclipse and the generation of JUnit code are examples. This paper is part of our effort to promote JCrasher for use in third-party development.

9. Acknowledgments

This work has been supported by the Yamacraw Foundation and by the NSF, under grants CCR-0220248 and CCR-0238289.

REFERENCES

1. Beizer B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* John Wiley & Sons, 1995.
2. Claessen K, Hughes J. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00),* Odersky M, Wadler P (eds.). ACM Press, 2000; 268–279.

3. Forrester JE, Miller BP. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows Systems Symposium*, 2000; 59–68.
4. Howe AE, von Mayrhauser A, Mraz RT. Test case generation as an AI planning problem. *Automated Software Engineering* 1997; **4**(1):77–106.
5. Kropp NP, Koopman PJ, Siewiorek DP. Automated robustness testing of off-the-shelf software components. In *Digest of Papers: FTCS-28, The Twenty-Eigth Annual International Symposium on Fault-Tolerant Computing*. IEEE Computer Society, 1998; 230–239.
6. Memon AM, Pollack ME, Soffa ML. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering* 2001; **27**(2):144–155.
7. Parasoft Jtest page. http://www.parasoft.com/jsp/products/home.jsp?product=Jtest [7 December 2003].
8. SilverMark Inc. Enhanced JUnit version 3.7 user reference. http://www.silvermark.com/Product/enhancedjunit/documentation/enhancedjunitmanual.pdf [7 December 2003].
9. Beck K, Gamma E. Test infected: programmers love writing tests. *Java Report* 1998; **3**(7):37–50.
10. Chan P, Lee R, Kramer D. 1998. *The Java Class Libraries* (2nd edn), vol. 1. Addison-Wesley, 1998.
11. Cheon Y, Leavens G. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP 2002 - Object-Oriented Programming, 16th European Conference,* Magnusson B (ed.). Springer, 2002; 231–255.
12. Dillenberger DN, Bordawekar R, Clark CW, Durand D, Emmes D, Gohda O, Howard S, Oliver MF, Samuel F, St.John RW. Building a Java virtual machine for server applications: the JVM on OS/390. *IBM Systems Journal* 2000; **39**(1):194–210.
13. Lindholm T, Yellin F. *The Java^{TM} Virtual Machine Specification* (2nd edn). Addison-Wesley, 1999.
14. Bytecode engineering library (BCEL) page. http://jakarta.apache.org/bcel/ [7 December 2003].
15. Kozen D, Stillerman M. Eager class initialization for Java. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium (FTRTFT 2002),* Damm W, Olderog ER (eds.). Springer, 2002; 71–80.
16. Stotts D, Lindsey M, Antley A. An informal formal method for systematic JUnit test case generation. In *XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference,* Wells D, Williams LA (eds.). Springer, 2002; 131–143.
17. Xie T, Notkin D. Tool-assisted unit test selection based on operational violations. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*. IEEE Computer Society, 2003; 40–48.
18. Junit Test Generator page. http://sourceforge.net/projects/junittestmaker/ [7 December 2003].
19. Pan J, Koopman P, Siewiorek D, Huang Y, Gruber R, Jiang ML. Robustness testing and hardening of CORBA ORB implementations. In *The International Conference on Dependable Systems and Networks (DSN'01)*. IEEE Computer Society, 2001; 141–150.
20. Edwards SH. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability* 2001; **11**(2):97–111.
21. Sankar S, Hayes R. ADL—an interface description language for specifying and testing software. *ACM SIGPLAN Notices* 1994; **29**(8):13–21.
22. Zweben SH, Heym WD, Kimmich J. Systematic testing of data abstractions based on software specifications. *Software Testing, Verification and Reliability* 1992; **1**(4):39–55.
23. Boyapati C, Khurshid S, Marinov D. Korat: automated testing based on Java predicates. In *ISSTA 2002, Proceedings of the International Symposium on Software Testing and Analysis,* Frankl PG (ed.). ACM Press, 2002; 123-133.
24. JCrasher page. http://www.cc.gatech.edu/~csallnch/jcrasher/ [7 December 2003].