

Mutation-driven Generation of Oracles and Unit Tests

Gordon Fraser, *Member, IEEE*, Andreas Zeller, *Member, IEEE*

Abstract—To assess the quality of test suites, *mutation analysis* seeds artificial defects (mutations) into programs; a non-detected mutation indicates a weakness in the test suite. We present an automated approach to generate unit tests that detect these mutations for object-oriented classes. This has two advantages: First, the resulting test suite is optimized towards finding *defects* rather than covering code. Second, the state change caused by mutations induces *oracles* that precisely detect the mutants. Evaluated on 10 open source libraries, our μ TEST prototype generates test suites that find significantly more seeded defects than the original manually written test suites.

Index Terms—Mutation analysis, test case generation, unit testing, test oracles, assertions, search based testing

1 INTRODUCTION

How good are my test cases? This question can be answered by applying *mutation analysis*: Artificial defects (mutants) are injected into software and test cases are executed on these fault-injected versions. A mutant that is not detected shows a deficiency in the test suite and indicates in most cases that either a new test case should be added, or that an existing test case needs a better test oracle.

Improving test cases after mutation analysis usually means that the tester has to go back to the drawing-board and design new test cases, taking the feedback gained from the mutation analysis into account. This process requires a deep understanding of the source code and is a non-trivial task. Automated test generation can help in covering code (and thus hopefully detecting mutants); but even then, the tester still needs to assess the results of the generated executions—and has to write hundreds or even thousands of oracles.

In this paper, we present μ TEST, an approach that automatically generates unit tests for object-oriented classes based on mutation analysis. By using mutations rather than structural properties as coverage criterion, we not only get guidance in *where* to test, but also *what* to test for. This allows us to generate effective *test oracles*, a feature raising automation to a level not offered by traditional tools.

As an example, consider Figure 1, showing a piece of code from the open source library *Joda Time*. This constructor sets `iLocalMillis` to the milliseconds represented by day, month, and year in the `values` array and 0 milliseconds. It is called by 12 out of 143 test cases for `LocalDate`—branches and statements are all perfectly covered, and each

```

1 public class LocalDate {
2   // The local milliseconds from 1970-01-01T00:00:00
3   private long iLocalMillis;
4   ...
5   // Construct a LocalDate instance
6   public LocalDate(Object instant, Chronology c) {
7     ... // convert instant to array values
8     ... // get iChronology based on c and instant
9     iLocalMillis = iChronology.getDateTimeMillis(
10      values[0],values[1],values[2],0); ← Change 0 to 1
11  }
12 }

```

Fig. 1. Mutating the initialization of `iLocalMillis` is not detected by the Joda Time test suite.

```

1 LocalDate var0 = new org.joda.time.LocalDate()
2 DateTime var1 = var0.toDateTimeAtCurrentTime()
3 LocalDate var2 = new org.joda.time.LocalDate(var1)
4 assertTrue(var2.equals(var0));

```

Fig. 2. Test generated by μ TEST. The call in Line 3 triggers the mutation; the final assertion detects it.

of these test cases also covers several definition-use pairs of `iLocalMillis`. These test cases, however, only check whether the day, month, and year are set correctly, which misses the fact that comparison between `LocalDate` objects compare the actual value of `iLocalMillis`. Consequently, if we mutate the last argument of `getDateTimeMillis` from 0 to 1, thus increasing the value of `iLocalMillis` by 1, this seeded defect is not caught.

μ TEST creates a test case with an oracle that catches this very mutation, shown in Figure 2. `LocalDate` object `var0` is initialized to a fixed value (0, in our case). Line 2 generates a `DateTime` object with the same day, month, and year as `var0` and the local time (fixed to 0, again). The constructor call for `var2` implicitly calls the constructor from Figure 1, and

• G. Fraser and A. Zeller are with the Chair of Software Engineering, Saarland University, Saarbrücken, Germany.
E-mail: fraser,zeller@cs.uni-saarland.de

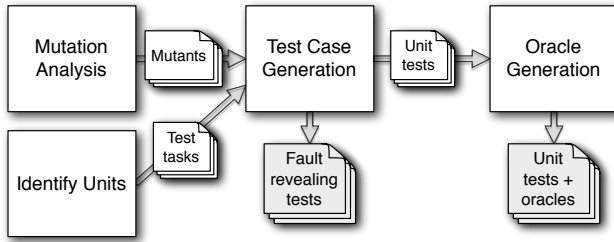


Fig. 3. The μ TEST process: Mutation analysis, unit partitioning, test generation, oracle generation.

therefore `var0` and `var2` have identical day, month, and year, but differ by 1 millisecond on the mutant, which the assertion detects.

By generating oracles in addition to tests, μ TEST simplifies the act of testing to checking whether the generated assertions are valid, rather than coming up with assertions and sequences that are related to these assertions. If a suggested assertion is not valid, then usually a bug has been found.

Summarizing, the contributions of this paper are:

Mutant-based oracle generation: By comparing executions of a test case on a program and its mutants, we generate a reduced set of assertions that is able to distinguish between a program and its mutants.

Mutant-based unit test case generation: μ TEST uses a genetic algorithm to breed method/constructor call sequences that are effective in detecting mutants.

Impact-driven test case generation: To minimize assessment effort, μ TEST optimizes test cases and oracles towards detecting mutations with maximal *impact*—that is, changes to program state all across the execution. Intuitively, greater impact is easier to observe and assess for the tester, and more important for a test suite.

Mutant-based test case minimization: Intuitively, the shorter a test case, the easier it is to understand. We minimize test cases by first preferring shorter sequences during test case generation, and then we remove all irrelevant statements in the final test case.

Figure 3 shows the overall μ TEST process: The first step is mutation analysis (Section 2) and a partitioning of the software under test into test tasks, consisting of a unit under test with its methods and constructors as well as all classes and class members relevant for test case generation. For each unit a genetic algorithm breeds sequences of method and constructor calls until each mutant of that unit, where possible, is covered by a test case such that its impact is maximized (Section 3). We minimize unit tests by removing all statements that are not relevant for the mutation or affected by it; finally, we generate and minimize assertions by comparing the behavior of the test case on the original software and its mutants (Section 4).

We have implemented μ TEST as an extension to the Javalanche [35] mutation system (Section 5). This paper extends an earlier version presented at IS-

STA 2010 [17], in which we used a first prototype to do preliminary analysis. In this paper, we use a matured version of μ TEST to extend the empirical analysis to a total of 10 open source libraries, making it one of the largest empirical studies of evolutionary testing of classes to date. The results show that our approach can be used to extend (and theoretically, replace) the manually crafted unit tests (Section 6).

2 BACKGROUND

2.1 Mutation Analysis

Mutation analysis was introduced in the 1970s [13] as a method to evaluate a test suite in how good it is at detecting faults, and to give insight into how and where a test suite needs improvement. The idea of mutation analysis is to seed artificial faults based on what is thought to be real errors commonly made by programmers. The test cases of an existing test suite are executed on a program version (mutant) containing one such fault at a time in order to see if any of the test cases can detect that there is a fault. A mutant that is detected as such is considered *dead*, and of no further use. A *live* mutant, however, shows a case where the test suite potentially fails to detect an error and therefore needs improvement.

There are two main problems of mutation analysis: First, the sheer number of mutants and the effort of checking all tests against all mutants can cause significant computational costs. Second, *equivalent* mutants do not observably change the program behavior or are even semantically identical, and so there is no way to possibly detect them by testing. Since the introduction of mutation analysis, a number of optimizations have been proposed to overcome possible performance problems, and heuristics can identify a small fraction of equivalent mutants—at the end of the day, however, detecting equivalent mutants is still a job to be done manually. A recent survey paper [21] concisely summarizes all applications and optimizations that have been proposed over the years.

Javalanche [35] is a tool that incorporates many of the proposed optimizations and made it possible to apply mutation analysis to software of previously unthinkable size. In addition, Javalanche alleviates the equivalent mutant problem by ranking mutants by their *impact*: A mutant with high impact is less likely to be equivalent, which allows the tester to focus on those mutants that can really help to improve a test suite. The impact can be measured, for example, in terms of violated invariants or effects on code coverage.

2.2 Test Case Generation

Research on automated test case generation has resulted in a great number of different approaches, deriving test cases from models or source code, using different test objectives such as coverage criteria,

and using many different underlying techniques and algorithms. In this paper, we only consider white-box techniques that require no specifications; naturally, an existing specification can help in both generating test cases and can serve as test oracle.

The majority of systematic white-box testing approaches consider the control-flow of the program, and try to cover as many aspects as possible (e.g., all statements or all branches). Generating test data by solving path constraints generated with symbolic execution is a popular approach (e.g., PathCrawler [44]), and dynamic symbolic execution as an extension can overcome a number of problems by combining concrete executions with symbolic execution. This idea has been implemented in tools like DART [18] and CUTE [37], and is also applied in Microsoft’s parametrized unit testing tool PEX [40] or the object-oriented testing framework Symstra [46].

Meta-heuristic search techniques have been used as an alternative to symbolic execution based approaches and can be applied to stateless and stateful programs [1], [26] as well as object-oriented container classes [6], [41], [43]. A promising avenue seems to be the combination of evolutionary methods with dynamic symbolic execution (e.g., [20], [23]), alleviating some of the problems both approaches have.

While systematic approaches in general have problems with scalability, random testing is an approach that scales with no problems to programs of any size. Besides random input generation a recent trend is also generation of random unit tests, as for example implemented in Randoop [32], JCrasher [12], AutoTest [11], or RUTE-J [3]. Although there is no guarantee of reaching certain paths, random testing can achieve relatively good coverage with low computational needs.

2.3 Test Case Generation for Mutation Testing

Surprisingly, only little work has been done on generating test cases dedicated to kill mutants, even though this is the natural extension of mutation analysis. DeMillo and Offutt [14] have adapted constraint based testing to derive test data that kills mutants. The idea is similar to test generation using symbolic execution and constraint solving, but in addition to the path constraints (called *reachability condition* by DeMillo and Offutt), each mutation adds a condition that needs to be true (*necessity condition*) such that the mutant affects the state. Test data is derived using constraint solving techniques again. Offutt et al. [28] also described an advanced approach to generate test data that overcomes some of the limitations of the original constraint based approach. Recently, a constraint based approach to derive test data for mutants has also been integrated in the dynamic symbolic execution based tool Pex [47].

Jones et al. [22] proposed the use of a genetic algorithm to find mutants in branch predicates, and

Bottacci [9] proposed a fitness function for genetic algorithms based on the constraints defined by DeMillo and Offutt [14]; recently this has been used for experiments using genetic algorithms and ant colony optimization to derive test data that kills mutants [7].

Differential test case generation (e.g., [15], [30], [39]) shares similarities with test case generation based on mutation analysis in that these techniques aim to generate test cases that show the difference between two versions of a program. Mutation testing, however, does not require the existence of two different versions to be checked and is therefore not restricted in its applicability to regression testing. In addition, the differences in the form of simple mutations are precisely controllable and can therefore be exploited for test case generation.

2.4 Oracle Generation

Automated synthesis of assertions is a natural extension of test case generation. Randoop [32] allows annotation of the source code to identify observer methods to be used for assertion generation. Orstra [45] generates assertions based on observed return values and object states and adds assertions to check future runs against these observations. A similar approach has also been adopted in commercial tools such as Agitar Agitator [8]. While such approaches can be used to derive efficient oracles, they do not serve to identify which of these assertions are actually useful, and such techniques are therefore only found in regression testing.

Eclat [31] can generate assertions based on a model learned from assumed correct executions; in contrast, μ TEST does not require any existing executions.

Evans and Savoia [15] generate assertions from runs of two different versions of a software system and DiffGen [39] extends the Orstra approach to generate assertions from runs on two different program versions. This is similar to our μ TEST approach, although we do not assume two existing different versions of the same class but generate many different versions by mutation, and are therefore not restricted to a regression testing scenario.

3 UNIT TESTS FROM MUTANTS

To generate test cases for mutants, we adopt an evolutionary approach in line with previous work on testing classes [6], [41]: In general, genetic algorithms evolve a population of chromosomes using genetics-inspired operations such as selection, crossover, and mutation, and each chromosome represents a possible problem solution.

To generate unit tests with a genetic algorithm, the first ingredient is a genetic representation of test cases. A unit test generally is a sequence of method calls on an object instance, therefore the main components are method and constructor calls. These methods and

constructors take parameters which can be of primitive or complex type. This gives us four main types of statements:

Constructor statements generate a new instance of the test class or any other class needed as a parameter for another statement:

```
DateTime var0 = new DateTime()
```

Method statements invoke methods on instances of any existing objects (or static methods):

```
int var1 = var0.getSecondOfMinute()
```

Field statements access public fields of objects:

```
DurationField var2 = MillisDurationField.  
    INSTANCE
```

Primitive statements represent numeric data types:

```
int var3 = 54
```

Each statement defines a new variable (except void method calls), and a chromosome is a sequence of such statements. The parameters of method and constructor calls may only refer to variables occurring earlier in the same chromosome. Constructors, methods, and fields are not restricted to the members of the class under test because complex sequences might be necessary to create suitable parameter objects and reach required object states.

The initial population of test cases is generated randomly as shown in Algorithm 1: First, we determine the set of all method and constructor calls that either contain a mutation or directly or indirectly call a mutated function. We randomly select one of these calls, and try to generate all parameters as well as its callee, if applicable. For this, we recursively either select existing objects or add new calls that yield the necessary objects (Algorithm 2). During this search, we keep track of classes we are already trying to initialize. If random generation turns out to be difficult for a particular class, Algorithm 2 can be turned into an exhaustive search by adding backtracking and keeping track of explored choices. Even though the search space can be big depending on the size of the software under test, this is not problematic as one can retain useful sequences that lead to creation of certain objects and reuse them later. Test cases are created in this way until the initial population has reached the required size.

Evolution of this population is performed by repeatedly selecting individuals from the current generation (for example, using tournament or rank selection) and applying crossover and mutation according to a certain probability. Figure 4 illustrates single point crossover for unit tests: A random position in each of the two selected parents is selected, and offspring is generated by trying to merge the sub-sequences. As statements can refer to all variables created earlier in the sequence this means simply attaching a cut-off

Algorithm 1 Random generation used for the initial population.

Require: C : Set of all methods and constructors directly/indirectly calling mutation

$t \leftarrow \langle \rangle$

$s \leftarrow$ random generator for unit under test

for callee and all parameters of s **do**

GENOBJECT(class of parameter, {}, t)

end for

$t \leftarrow t.s$

while $|t| <$ desired length **do**

$s \leftarrow$ random method using an object from t as either callee or parameter

for callee and all parameters of s not existing in t **do**

GENOBJECT(class of parameter, {}, t)

end for

$t \leftarrow t.s$

end while

return t

Algorithm 2 GENOBJECT: Recursive generation of objects.

Require: c : Class of desired object

Require: G : Set of classes already attempting to generate

Require: t : Test case

if t has object of class c **then**

return existing object with certain probability

end if

$s \leftarrow$ random element from all generators

if $s =$ non-static method or field **then**

$G \leftarrow G \cup \{c\}$

Set callee of s to GENOBJECT(Class of callee, G , t)

end if

for all parameters of s **do**

Set parameter to GENOBJECT(Class of parameter, G , t)

end for

$t \leftarrow t.s$

return t

sub-sequence will not work. Instead we add the statements and try to satisfy parameters with the existing variables, or possibly create new objects to satisfy all parameters, similarly to the initial generation (Algorithm 1). By choosing different points of crossover in the two parent chromosomes the crossover will result in a variation of the length of test cases.

After selection and crossover the offspring is mutated with a given probability. There are numerous ways to mutate chromosomes:

Delete a statement: Drop one random statement in the test case. The return value of this statement must not be used as a parameter or callee in

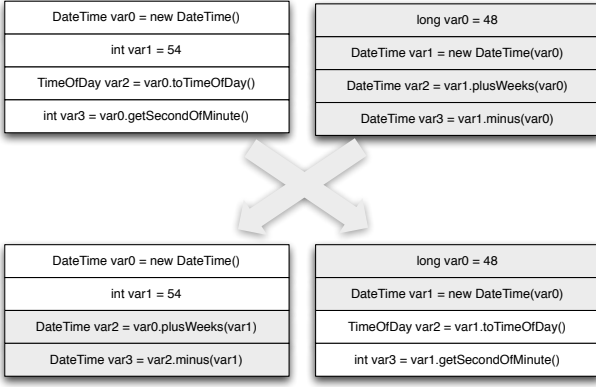


Fig. 4. Crossover between two test cases.

another statement, unless it can be replaced with another variable.

Insert a method call: Add a random method call for one of the objects of the test case, or create a new object of a type already used in the test case.

Modify an existing statement: Apply one of the following changes on one of the existing statements:

- *Change callee:* For a method call or field reference, change the source object.
- *Change parameters:* Change one parameter of a method or constructor call to a different value or create new value.
- *Change method/constructor:* Replace a call with a different one with identical return type.
- *Change field:* Change a field reference to a different field of the same type on the same class.
- *Change primitive:* Replace a primitive value with a different primitive value of the same type.

When mutating a chromosome, with a certain probability each statement is changed or deleted, or a new statement is inserted before it.

In order to guide the selection of parents for offspring generation, all individuals of a population are evaluated with regard to their fitness. The fitness of a test case with regard to a mutant is measured with respect to how close it comes to (1) executing the mutated method, (2) the mutated statement in this method, and (3) how significant the impact of the mutant on the remaining execution is. Consequently, the fitness function is a combination of these three components:

Distance to calling function

If the method/constructor that contains the mutation is not executed, then the fitness represents the distance towards achieving this, i.e., how many of the necessary parameters and callees are generated for a method that directly or indirectly executes the mutation. Generally, we want the distance to be as small as possible. This value can be determined without executing the test case, and is 1 if the mutant is

executed. We define this distance as a function of a test case t :

$$d(c, t) := 1 + \# \text{ Unsatisfied parameters} \quad (1)$$

$$D_f(t) := \min\{d(c, t) \mid \text{calls } c \text{ related to mutation}\} \quad (2)$$

If the target method is not static, then the callee object also counts as a parameter that needs to be satisfied.

Distance to mutation

If the mutant method/constructor is executed but the mutated statement itself is not, then the fitness specifies the distance of the test case to executing the mutation; again, we want to minimize this distance. This basically is the prevailing *approach level* and *branch distance* measurement applied in search-based test data generation [26]: The approach level describes the number of unsatisfied control dependencies, i.e., how many branches need to evaluate differently to reach the mutation. The branch distance estimates how far the branch at which execution diverged from reaching the mutation is from evaluating to the necessary outcome. In addition, one can use the necessity conditions [14] definable for different mutation operators to estimate the distance to an execution of the mutation that infects the state (necessity distance [9]). To determine this value the test case has to be executed once on the unmodified software; if the mutation is properly executed then the value will be 0. As usual, the branch distance should not dominate the approach level, and therefore it is normalized, for example using the normalization function $\alpha(x) = \frac{x}{x+1}$ [4].

$$D_m(t) := \text{Approach} + \alpha(\text{Branch/Necessity Distance}) \quad (3)$$

Mutation impact

If the mutation is executed, then we want the test case to propagate any changes induced by the mutation such that they can be observed. Traditionally, this consists of two conditions: First, the mutation needs to infect the state (necessity condition). This condition can be formalized and integrated into the fitness function (see above). In addition, however, the mutation needs to propagate to an observable state (sufficiency condition) — it is difficult to formalize this condition [14]. Therefore, we measure the *impact* of the mutation on the execution; we want this impact to be as large as possible.

Quantification of the impact of mutations was initially proposed using dynamic invariants [34]: The more invariants of the original program a mutant program violates, the less likely it is to be equivalent. A variation of the impact measurement uses the number of methods with changed coverage or return

values [36] instead of invariants. We take a slightly different view on the impact, as strictly speaking it is not just a function of the mutant itself, but also of the test case. Consequently, we use the impact as part of the fitness function, and quantify the impact of a mutation as the unique number of methods for which the coverage changed and the number of observable differences. An observable difference is, for example, a changed return value or any other traceable artifact (cf. Section 4). Using the unique number of changed methods avoids ending up with long sequences of the same method call.

$$I_m(t) = c \times |C| + r \times |A| \quad (4)$$

Here, C is the set of called statements with changed coverage, and A is the set of observable differences between a run on the original and the mutant program; c and r are constants that allow to put more weight on observable changes. To determine the impact, a test case has to be executed on the original unit and on the mutant that is currently considered.

Overall fitness

The overall fitness function is a combination of these three factors; the two distances have to be minimized, while the impact should be maximized. As long as $D_f(t) > 1$ and $D_m(t) > 0$, $I_m(t)$ will be 0 per definition. Consequently, a possible combination as a fitness function that should be maximized is as follows:

$$fitness(t) = \frac{1}{D_f(t) + D_m(t)} + I_m(t) \quad (5)$$

The fitness is less than 1 if the mutant has not been executed, equal to 1 if the mutant is executed but has no impact, and greater than one if there is impact or any observable difference.

Individuals with dynamic length suffer from a problem called *bloat*, which describes an unproportional growth of the individuals of a population. In contrast, from a user perspective test cases should be as short as possible to allow easier understanding. In our initial prototype [17] we integrated the length directly into the fitness function. This, however, might have negative side-effects. Consequently, following the insights of our recent extensive experimentation [16] we do not include the length explicitly in the fitness function, but implicitly in terms of the rank selection: If two individuals have the same fitness, the shorter one will be ranked before the longer individual.

When generating test cases for structural coverage criteria the stopping criterion is easy: If the entity to be reached (e.g., a branch) is actually reached, then one is done. In our scenario this is not so easy: It might not be sufficient to stop once a mutant has resulted in an observable difference, as we are also

optimizing with respect to the test case length and the impact. Consequently, one may choose to let the genetic algorithm continue until a maximum time or number of generations has been reached, and returns the test case with the highest fitness.

As determining the fitness value of a test case requires it to be executed, a nice side-effect is that memory violations and uncaught exceptions are already detected during test case generation. If such exceptions are not declared to occur, then they are likely to point to defects (see Section 5).

4 GENERATING ASSERTIONS TO KILL MUTANTS

The job of generating test cases for mutants is not finished once a mutation is executed: A mutant is only detected if there is an oracle that can identify the misbehavior that distinguishes the mutant from the original program. Consequently, mutation-based unit tests need to add assertions as test oracles such that the mutants are detected.

Some types of assertions (e.g., assertions on primitive return values) can be easily generated, as demonstrated by existing testing tools (cf. Section 2.4). This, however, is not so easy for all types of assertions, and the number of possible assertions in a test case typically exceeds the number of statements in the test case by far. Mutation testing helps in precisely this matter by suggesting not only where to test, but also what to check for. This is a very important aspect of mutation testing, and has hitherto not received attention for automated testing.

After the test case generation process we run each test case on the unmodified software and all mutants that are covered by the test case, while recording traces with information necessary to derive assertions. We use the following types of assertions, all illustrated with examples of actually generated test cases:

Primitive assertions make assumptions on primitive (i.e., numeric or Boolean) return values:

```
DurationField var0 =
    MillisDurationField.INSTANCE;
long var1 = 43;
long var2 = var0.subtract(var1, var1);
assertEquals(var2, 0);
```

Comparison assertions compare objects of the same class with each other. Comparison of objects across different execution traces is not safe because it might be influenced by different memory addresses, which in turn would influence assertions; therefore we compare objects *within* the same test case. For classes implementing the Java `Comparable` interface we call `compareTo`; otherwise, we apply comparison to all objects of compatible type in terms of the Java `equals` method:

```
DateTime var0 = new DateTime();
Chronology var1 = Chronology.getCopticUTC();
DateTime var2 = var0.toDateTime(var1);
assertFalse(var2.equals(var0));
```

Inspector assertions call inspector methods on objects to identify their states. An inspector method is a method that takes no parameters, has no side-effects, and returns a primitive data type. Inspectors can, for example, be identified by purity analysis [33].

```
long var0 = 38;
Instant var1 = new Instant(var0);
Instant var2 = var1.plus(var0);
assertEquals(var2.getMillis(), 76);
```

Field assertions are a variant of inspector assertions and compare the public primitive fields of classes among each other directly. (As Joda Time has no classes with public fields, there is no example from this library).

String assertions compare the string representation of objects by calling the `toString` method. For example, the following assertion checks the ISO 8601 representation of a period in Joda Time:

```
int var0 = 7;
Period var1 = Period.weeks(var0);
assertEquals("P7W", var1.toString());
```

String assertions are not useful in all cases: First, they are only usable if the class implements this method itself, as the `java.lang.Object.toString` method inherited by all classes includes the memory location of the reference—which does not serve as a valid oracle. Second, the `toString` method is often used to produce debug output which accesses many internals that might else not be observable via the public interface. Oracles depending on internal details are not proof to changes in the code, and we therefore only use these assertions if no memory locations are included in the string and if no other assertions can be generated.

Null assertions compare object references with the special value `null`:

```
int var0 = 7;
Period var1 = Period.weeks(var0);
assertNonNull(var1);
```

Exception assertions check which exceptions are raised during execution of a test case:

```
@Test (expected=IllegalArgumentException.class)
int var0 = -20;
DateTime var1 = new DateTime(var0,var0,var0); //
    should raise exception
```

To generate assertions for a test case we run it against the original program and all mutants using

observers to record the necessary information: An observer for primitive values records all observed return and field values, while an inspector observer calls all inspector methods on existing objects and stores the outcome, and a comparison observer compares all existing objects of equal type and again stores the outcome. After the execution the traces generated by the observers are analyzed for differences between the runs on the original program and its mutants, and for each difference an assertion is added. At the end of this process, the number of assertions is minimized by tracing for each assertion which mutation it kills, and then finding a subset for each test case that is sufficient to detect all mutations that can be detected with this test case. This is an instance of the NP-hard minimum set covering problem, and we therefore use a simple greedy heuristic [10]. The heuristic starts by selecting the best assertion, and then repeatedly adds the assertion that detects the most undetected mutants. To give preference to other types of assertions, string assertions are only added at the end of this procedure to cover those mutants that can only be covered by string assertions.

5 GENERATING JAVA UNIT TEST SUITES

Being able to generate unit tests and assertions gives us the tools necessary to create or extend entire test suites. We have implemented the described μ TEST approach as an extension to the Javalanche [35] mutation system.

5.1 Mutation Analysis

The first step of a mutation based approach is to perform classic mutation analysis, which Javalanche does efficiently: Javalanche instruments Java bytecode with mutation code and runs JUnit test cases against the instrumented (mutated) version. The result of this process is (a) a classification of mutants into dead or live with respect to the JUnit test suite, as well as (b) an impact analysis on live mutants that have been executed but not detected.

5.2 Setup

The second step on the way to a test suite is to extract testable units with all necessary information:

- Mutations of the unit under test (UUT)
- Testable constructors and methods of the UUT
- Classes and their members necessary to execute all methods and constructors of the UUT
- Inspector methods of the UUT

During test case generation, μ TEST accesses the UUT via Java reflection, but at the end of the test case generation process the test cases are output as regular JUnit test cases. Therefore, an important factor for the testability of a Java class is its accessibility. This means that all public classes can be tested, including

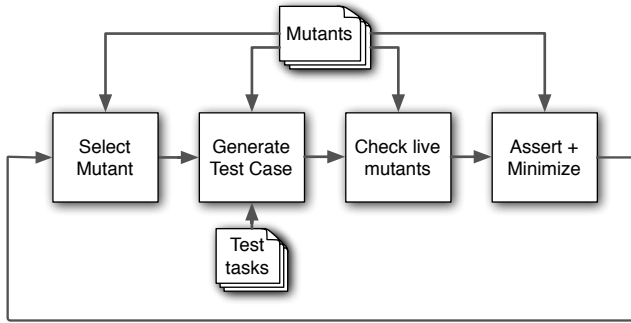


Fig. 5. The process of generating test cases for one unit.

public member classes. Private or anonymous classes, on the other hand, cannot be explicitly tested but only indirectly via other classes that access them. Of the accessible classes, all constructors, methods, and fields declared public can be accessed. As abstract classes cannot be instantiated directly, we consider all derived subclasses when deriving information about testable members for a class.

A basic requirement to test methods and constructors is the ability to create objects of all parameter types; therefore each parameter type is analyzed whether it offers all necessary generators to be applicable to Algorithm 2. Alternatively, the user can add helper functions that create usable objects, or theoretically one could also keep a pool of created objects [11]. A further requirement for a method to be testable is that it can be executed repeatedly during the search, and that calling the same method with identical parameters always returns the same result (i.e., it is deterministic).

For each unit, we only consider the mutants derived at this level in the type hierarchy, i.e., for an abstract class only mutants of that class are used although members of subclasses are necessary to derive test cases. The intuition of this is that it will be easier for the tester to understand test cases and assertions if she only has to focus on a single source file at a time.

In addition to the test methods, each unit is analyzed with respect to its inspector methods. An inspector method is a method that does not change the system state but only reports some information about it, and is therefore very useful for assertion generation. For Java, one has to resort to purity analysis [33] or manual input to identify inspectors.

5.3 Test Case Generation

Once the setup is finished, the actual test case generation can be started, possibly in parallel for different units. μ TEST selects a target mutant and generates a test case for it (see Figure 5). To measure the fitness of individuals, μ TEST executes test cases using Java reflection. The resulting test case is checked against all remaining live mutants to see if it also kills

other mutants. For all killed mutants, we derive a minimized version of the test case, enhanced with assertions, and add it to the test suite. The aim of minimization is to make the test case simpler and remove any unnecessary statements. For this we use a rigorous approach and try to remove each statement from the test case and check the test case’s validity and fitness afterward. Although this is basically a costly approach, it is not problematic in practice, as the rank selection favors short test cases, and they are therefore usually short to begin with. Alternatively, one could for example apply delta-debugging [24] or a combination of slicing and delta-debugging [25] to speed up minimization. This process is repeated until all mutants are killed or at least test case generation has been attempted on each of them.

5.4 Output

At the end of the process, the tester receives a JUnit test file for each unit, containing test cases with assertions and information about which test case is related to which mutant. Unless the test cases are used for regression testing, the assertions need to be analyzed and confirmed by the tester, and any assertion that is not valid reveals a bug in the software, or an invalid test input due to an implicit constraint that needs to be declared. To aid the developer, the number of assertions is minimized by only including enough assertions to kill all possible mutants.

Besides test cases, the developer also receives information about the mutants considered during test case generation: If one or more assertions could be generated, the mutant is considered to be killed. If the mutant was executed but no assertions could be found, then the impact is returned, such that mutations can be ranked according to their probability of being inequivalent. Some mutants cannot be tested (such as mutations in private methods that are not used in public methods), and some mutants are not supported by the tool itself (for example, mutations in static class constructors, as this would require unloading the class at each test run). Finally, a mutant might simply not have been executed because the tool failed to find a suitable sequence.

In addition to bugs revealed by checking assertions, the search for test cases itself exercises the UUT in random ways, thus possibly detecting defects commonly found with random testing (e.g., null pointer references and other uncaught exceptions.) Only undeclared exceptions are reported to the user, as an exception that is declared to occur in the source code is acceptable behavior. An undeclared exception, on the other hand, is either caused by a real bug in the UUT, or by implicit assumptions on method parameters or call sequences.

If the developer decides that an exception is not caused by a bug in the implementation, the test case

TABLE 1
Statistics on case study subjects

Case Study	Classes		Mutants	
	Testable	Total	Testable	Total
Commons CLI	13	20	652	1,276
Commons Codec	20	29	3,075	4,041
Commons Collections	193	273	13,188	21,705
Commons Logging	10	14	286	1,317
Commons Math	243	388	25,226	43,273
Commons Primitives	154	238	2,385	5,734
Google Collections	83	131	4,054	11,339
JGraphT	112	167	2,341	5,139
Joda Time	123	154	11,163	23,145
NanoXML	1	2	614	954
Σ	952	1416	62,984	117,913

generation tool can be told to ignore exceptions of this kind in the future, or write a partial specification for the offending method, e.g., constraints on value ranges for numeric parameters, or constraints on the types of classes used, and so on.

6 EXPERIMENTS

To learn about the applicability and feasibility of the presented approach, we applied μ TEST to a set of 10 open source libraries.

6.1 Case Study Subjects

In selecting case study subjects for a thorough evaluation of the approach we tried to not only use container classes, which are mainly used in the literature. There are, however, limitations to automated test case generation, and we therefore had to select open source projects that do not rely on networking, I/O, multithreading, or GUIs.

Table 1 summarizes statistics of the case study subjects. The number of classes listed in this table represents only top-level classes (i.e., no member classes). Not all classes are suitable for test generation, as to produce executable JUnit test cases the classes need to be publicly accessible and Javalanche needs to be able to create mutants; we also excluded exception classes.

Commons CLI (CLI)

The Apache Commons CLI¹ library provides an API for parsing command line options.

Commons Codec (CDC)

Commons Codec² provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs.

1. <http://commons.apache.org/cli/>
2. <http://commons.apache.org/codec/>

Commons Collections (COL)

Commons Collections³ is a collection of data structures that seek to improve over Java standard library classes. Commons collections makes heavy use of Java Generics, which allow to parametrize classes and methods with types. Unfortunately, this type information is removed from instances of generic classes by the compiler and therefore not accessible at runtime by reflection, and so in most cases a type parameter is just as difficult to treat as a parameter of type `Object`. For this case study subject, we used `Integer` objects whenever a method has an `Object` in its signature, as integer numbers are easy to generate and have a defined ordering.

Commons Logging (LOG)

The Commons Logging⁴ package bridges between different popular logging implementations, which can also be changed at runtime.

Commons Math (MTH)

Commons Math⁵ is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language.

Commons Primitives (PRI)

The Commons Primitives⁶ library contains a set of datastructures and utilities that are optimized for primitive datatypes.

Google Collections (GCO)

The Google Collections⁷ library is a set of new collection types and implementations.

JGraphT (JGT)

GraphT is a graph library that provides mathematical graph-theory objects and algorithms.⁸

Joda Time (JOT)

Joda Time⁹ provides replacement for the Java date and time classes. Joda Time is known for its comprehensive set of developer tests, providing assurance of the library's quality, which makes it well suited as a benchmark to compare automatically generated tests with. To make sure that test cases and generated assertions are deterministic, we configured the global `SystemMillisProvider` to return a constant value.

NanoXML (XML)

NanoXML¹⁰ is a small XML parser for Java. As the parser framework relies heavily on I/O to read, transform, and write XML files we only tested the NanoXML Lite version.

3. <http://commons.apache.org/collections/>
4. <http://commons.apache.org/logging/>
5. <http://commons.apache.org/math/>
6. <http://commons.apache.org/primitives/>
7. <http://code.google.com/p/google-collections/>
8. <http://www.jgrapht.org/>
9. <http://joda-time.sourceforge.net/>
10. <http://devkix.com/nanoxml.php>

6.2 Experimental Setup

Test case generation only requires Java bytecode, and we used the μ TEST prototype “out of the box” as far as possible, i.e., without writing test helper functions or modifying the software under test. Minor adaptations were only necessary to avoid calling methods with nondeterministic results like random number generators; these methods are easily detectable as generated assertions will not always hold on the original program either.

For all libraries we generated complete test suites with μ TEST, and compared them with the already existing unit tests. We configured μ TEST to use a steady state genetic algorithm with a population size of 100, and an elitism size of 1 individual. The maximum test case length was set to 60 statements, and evolution was performed for a maximum of 1,000,000 executed statements for each class. Before starting the evolution, we ran 100 random test cases on each class to identify trivial mutants, to focus the search on the difficult objectives. For the remaining mutants, the budget of statements was equally divided. For each mutant, the search was ended as soon as an observable difference was found, i.e., we did not optimize the impact further. In addition to the statement limit, we used a timeout of 10 minutes for each individual class, as some of the more complex classes can take very long, in particular if mutations lead to infinite loops. We used a timeout of 5 seconds for individual tests, and if a mutant timed out where the same test case run on the original program did not timeout we assume that the mutant causes an infinite loop and end the search for this particular mutant at this point.

Crossover probability was set to 75%, and for every offspring of length n each statement had a mutation probability of $1/n$. Mutation was performed with probability $1/3$ as insertion, $1/3$ as deletion, and $1/3$ as change. As μ TEST uses randomized algorithms, each experiment was repeated 30 times with different random seeds.

6.3 Results

Figure 6 shows the achieved mutation scores for each of the case study subjects. The mutation score is the ratio of killed mutants to mutants in total (including equivalent mutants, as we do not know which mutants are equivalent). NanoXML poses problems for μ TEST, as it requires XML input which is unlikely to be produced automatically. Similarly, Commons Logging hit the boundaries of what is possible with automated test generation. For the other projects, μ TEST was able to produce test suites with high mutation scores. Figure 7 illustrates that the number of test cases resulting per class is small on average; this is not surprising, as each test case not only kills the target mutation for which it was created, but also many other mutants.

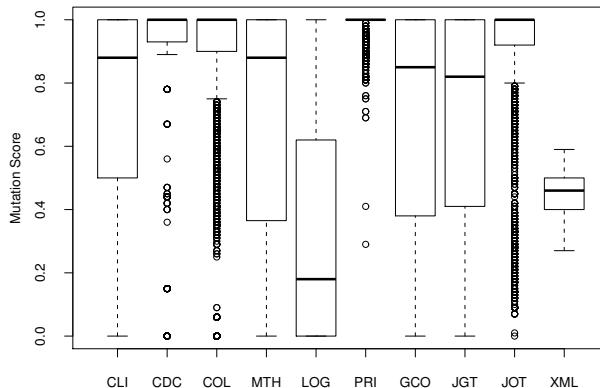


Fig. 6. Mutation score per class: For the majority of case study subjects and classes, μ TEST achieves high mutation scores.

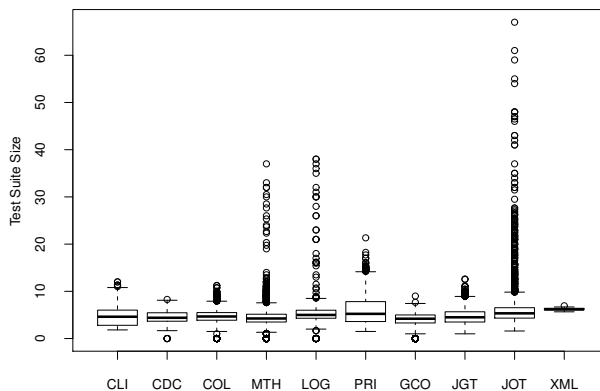


Fig. 7. Number of test cases in a test suite for a single class: Usually, few test cases are necessary to kill all mutants of a class.

We define the length of a test case as the number of statements it consists of, excluding assertions. The test suite length is the sum of the lengths of its constituent test cases. Figure 8 lists the test suite lengths, confirming that the resulting test suites are small and the minimization is effective. This is essential, as the test suites need to be presented to a developer who needs to confirm the generated oracles (except in a regression testing scenario). Small test suites with short test cases are important to allow the developer to understand the tests, and the oracles also contribute to the understandability as they serve as explanations of what the test case checks for.

Table 2 gives more detailed statistics on the hand-crafted test suites provided by the case study subjects as well as those generated by μ TEST. For the hand-crafted tests, we derived these numbers by counting

TABLE 2
Total number of test cases and average statistics per test case: Manually handcrafted vs. generated

Case Study	Manual			Generated		
	Tests	Statements/Test	Assertions/Test	Tests	Statements/Test	Assertions/Test
Commons CLI	187	7.45	2.80	137.39	4.91	2.57
Commons Codec	284	6.67	3.16	236.28	4.50	1.20
Commons Collections	12,954	6.28	2.10	1955.67	4.65	2.24
Commons Logging	26	6.90	1.03	77.86	6.08	2.00
Commons Math	14,693	6.93	3.41	1797.79	4.49	1.91
Commons Primitives	3,397	4.05	0.86	1145.67	5.88	1.54
Google Collections	33,485	4.52	1.25	781.79	3.88	1.81
JGraphT	118	9.10	1.65	484.96	4.56	1.52
Joda Time	3,493	4.89	4.55	1553.36	6.10	1.89
NanoXML	2	12.67	0.67	35.47	6.22	1.13

TABLE 3

\hat{A}_{12} measure values in the mutation score comparisons: $\hat{A}_{12} < 0.5$ means μ_{TEST} achieved lower, $\hat{A}_{12} = 0.5$ equal, and $\hat{A}_{12} > 0.5$ higher mutation score than the manually written test suites.

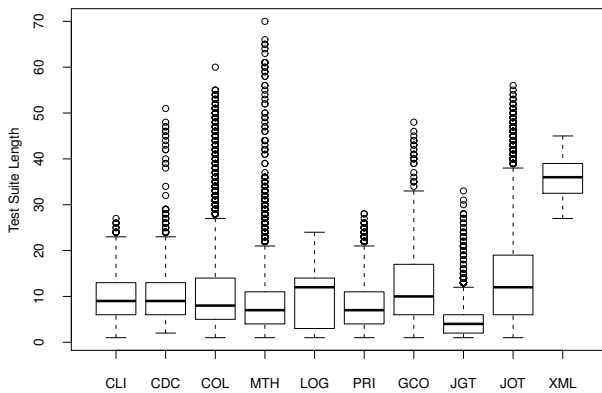


Fig. 8. Total length of a test suite for a single class: Short test cases are important to keep test cases understandable.

the number of non-commenting source code statements using JavaNCSS¹¹, and we counted the number of assertions using the command line tools `grep` and `wc`. The number of statements in Table 2 is derived as the number of non-commenting source code statements excluding assertions. As some of the subjects use parametrized tests, where one test method results in several unit tests, we averaged these numbers over the number of methods as reported by JavaNCSS. NanoXML Lite only comes with two test cases, but these tests parse and act upon complex XML files.

Unlike our initial results [17] the automatically generated test cases are on average shorter than manually written test cases. This demonstrates the effectiveness of both the genetic algorithm in finding short sequences and the minimization applied in removing all unnecessary statements. In only three out of 10 cases μ_{TEST} produced more test cases than in the manual test suites. To some extent, this is because we only considered the automatically testable subset of

Case Study	$\#\hat{A}_{12} < 0.5$	$\#\hat{A}_{12} = 0.5$	$\#\hat{A}_{12} > 0.5$
Commons CLI	7	1	5
Commons Codec	9	2	9
Commons Collections	46	24	123
Commons Logging	1	2	1
Commons Math	74	10	159
Commons Primitives	7	96	51
Google Collections	36	9	38
JGraphT	30	16	66
Joda Time	42	3	78
NanoXML	1	0	0
Σ	253	163	530

classes, while the manually written test cases address all classes.

Another explanation is that the automatically generated test cases simply are more effective at finding mutants than manually written test cases. To verify this explanation, we analyzed and compared the test suites for the testable classes in detail. As randomized algorithms can produce different results in different runs, we follow the guidelines on statistical analysis described by Arcuri and Briand [5].

Statistical difference has been measured with the Mann-Whitney U test. To quantify the improvement in a standardized way, we used the Vargha-Delaney \hat{A}_{12} effect size [42]. In our context, the \hat{A}_{12} is an estimation of the probability that, if we run μ_{TEST} , we obtain a higher mutation score than the manually crafted tests. When the two types of tests are equivalent, then $\hat{A}_{12} = 0.5$. A high value $\hat{A}_{12} = 1$ would mean that μ_{TEST} obtained a higher mutation score in *all* cases.

Table 3 shows how many times we obtained \hat{A}_{12} values equal, lower and higher than 0.5. We obtained p-values lower than 0.05 in 709 out of 946 comparisons. Figure 9 shows a box-plot of the results of

11. <http://www.kclee.de/clemens/java/javancss>

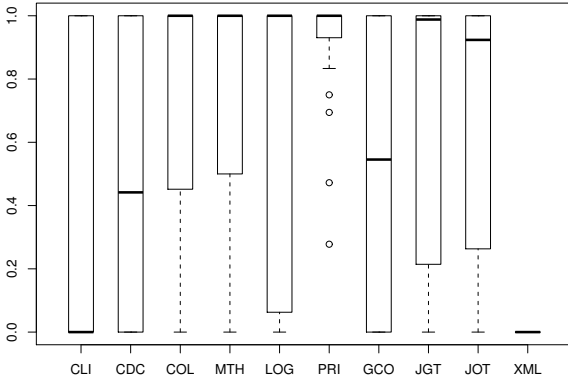


Fig. 9. \hat{A}_{12} for mutation scores: μTEST has a high probability of achieving a higher mutation score than manually written test cases.

the $\hat{A}_{12} \neq 0.5$ measure for the coverage grouped by case study subject; this figure illustrates the statistical evidence that μTEST achieves higher mutation scores.

μTEST generates test suites and oracles that find significantly more seeded defects than manually written test suites.

Considering that the test cases produced by μTEST are more effective at finding mutants, it seems counterintuitive that the number of test cases is so much smaller, and the tests are shorter (cf. Table 2). Are manually written test cases really that inefficient?

There is a danger that the test cases produced by μTEST are overfitted to the mutants that are considered for test case generation. To examine this effect, we performed two types of analysis: First, we analyzed the effectiveness of μTEST 's test cases to find defects that are not directly in the testable classes, i.e., the entire case study subject. Second, as a source of independent faults, we used an additional mutation operator to seed additional faults. By default, Javalanche uses the mutation operators *statement deletion*, *jump negation*, *arithmetic operator replacement*, and *comparison operator replacement*. Javalanche was recently extended with an additional mutation operator, *variable replacement* (VRO), which according to previous experiments [29], [38] should not be subsumed by the other operators. Consequently, we evaluated the test suites produced by μTEST with respect to their effectiveness at finding variable replacement faults.

Table 4 shows that in five out of 10 cases the mutation score on the standard operator set is higher for μTEST 's test suites than for the manually written test cases, and in the other five it is worse. One

12. Not all tests of Commons Logging are executable within Javalanche; the mutation score therefore only represents the score of those mutants for which the test cases executed successfully.

TABLE 4
Mutation scores for entire case study subjects, including all classes.

Case Study	Manual		Generated	
	Total	VRO	Total	VRO
Commons CLI	84.48%	81.85%	79.52%	32.45%
Commons Codec	85.99%	90.80%	97.45%	80.67%
Commons Collections	69.04%	62.44%	76.91%	47.50%
Commons Logging	72.14% ¹²	85.71% ¹¹	18.91%	10.00%
Commons Math	48.80%	53.53%	58.61%	16.24%
Commons Primitives	81.95%	76.41%	95.97%	54.27%
Google Collections	49.09%	57.96%	63.52%	45.04%
JGraphT	68.32%	65.22%	64.15%	23.52%
Joda Time	65.83%	67.90%	65.35%	48.94%
NanoXML	63.42%	19.14%	45.85%	45.62%
Average	69.75%	66.10%	68.38%	40.43%

conclusion of this observation is that μTEST 's ability to kill mutants depends on the testability of the software under test. For example, classes that need textual input, files, etc. are difficult to test. We can also see that the smaller case study subjects tend to be better tested by the developers than large projects, such as Commons Math. This demonstrates that μTEST would be perfectly suited to assist the developer in coming up with *additional* tests to cover more potential faults.

On the variable replacement operator (VRO) the manually written test cases achieve very similar results as on the other operators (see Table 4). In contrast, μTEST 's test cases show significantly lower mutation scores for this operator. This indicates that μTEST indeed overfits the test cases to the mutants considered during test case generation. However, we argue that this is not a problem: As was extensively determined experimentally [29], [38], there are *sufficient sets* of mutation operators: If a test suite kills all mutants derived with such a sufficient set, then it will detect all other mutants with very high probability. Just like mutation is an appropriate tool for testing experiments [2], it can also be a suitable tool to drive test generation. Given the competent programmer hypothesis and the coupling effect [27], such a test suite should detect most real faults. The addition of the variable replacement operator gives Javalanche and μTEST such a sufficient set of mutation operators.

6.4 Threats to Validity

The results of our experiments are subject to the following threats to validity:

- Our 952 classes investigated come from 10 projects; while this is a large case study with respect to the state of the art and the literature, we cannot claim that the results of our experimental evaluation are generalizable. The evaluation should be seen as investigating the potential of the technique rather than providing a statement of general effectiveness. The unit tests against

which we compared might not be representative for all types of tests (in particular we did not compare against automatically derived tests, as other tools do not provide oracles or only regression oracles), but we chose projects known to be well tested. The units investigated had to be testable automatically as discussed in Sections 5.2 and 6.1. The generalization to other programming languages than used in our experiments (Java) depends on the testability—for example, a search based approach as used by μ TEST would be quite hopeless on an untyped language such as Python, unless putting in a major effort in type analysis. The results should, however, generalize to comparable languages such as C# or C++.

- Another possible threat is that the tools we have used or implemented could be defective. To counter this threat, we have run several manual assessments and counter-checks.
- We evaluated the quality of unit tests in terms of their mutation score (Table 2). We also showed that the generated oracles may be overfitted to the mutants for which they are constructed, but studies (e.g., [2]) have shown that mutation analysis is well suited to evaluate testing techniques. In practice other factors such as understandability or maintainability have to be considered as well. To this extent, we try to maximize impact and minimize the length of test cases.

7 CONCLUSIONS

Mutation analysis is known to be effective in evaluating existing test suites. In this paper, we have shown that mutation analysis can also *drive automated test case generation*. The main difference between using structural coverage and mutation analysis to guide test case generation is that a mutation does not only show *where* to test, but also helps in identifying *what* should be checked for. In our experiments, this results in test suites that are significantly better in finding defects than the (already high-quality) manually written test suites.

The advent of automatic generation of effective test suites has an impact on the entire unit testing process: Instead of laboriously thinking of sequences that lead to observable features and creating oracles to check these observations, the tester lets a tool create unit tests for her automatically, and receives two test sets: One revealing general faults detectable by random testing, the other one consisting of regular unit tests. In the long run, finding bugs could thus be reduced to the task of checking whether the generated assertions match the intended behavior.

Although our μ TEST experiences are already very promising, there is ample opportunity to improve the results further: For example, previously generated test cases, manual unit tests, or test cases satisfying a coverage criterion could serve as a better starting point

for the genetic algorithm. The search based algorithm can be much optimized, for example by applying testability transformation [19], or improving the fitness function. If a mutated method is executed but the mutant is not executed, then a local optimization on the parameters of that method call could possibly lead to a result much quicker than the global search (for example, if the input parameters are numeric). It is even conceivable to use a hybrid approach with dynamic symbolic execution for this task.

To learn more about our work on mutation testing, visit our Web site:

<http://www.st.cs.uni-saarland.de/mutation/>

ACKNOWLEDGMENTS

We thank David Schuler for his work on Javalanche and his patience and support in extending Javalanche, Andrea Arcuri for his wisdom on search-based testing and statistical analysis, and Valentin Dallmeier, Andrzej Wasylkowski, Jeff Offutt, Eva May, and the anonymous reviewers for comments on earlier versions of this paper.

REFERENCES

- [1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2009.
- [2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 402–411, New York, NY, USA, 2005. ACM.
- [3] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li. Tool support for randomized unit testing. In *RT '06: Proceedings of the 1st International Workshop on Random Testing*, pages 36–45, New York, NY, USA, 2006. ACM.
- [4] A. Arcuri. It does matter how you normalise the branch distance in search based software testing. In *ICST'10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 205–214. IEEE Computer Society, 2010.
- [5] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *IEEE International Conference on Software Engineering (ICSE)*, 2011.
- [6] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.
- [7] K. Ayari, S. Bouktif, and G. Antoniol. Automatic mutation test input data generation via ant colony. In *GECCO '07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, pages 1074–1081, New York, NY, USA, 2007. ACM.
- [8] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 169–180, New York, NY, USA, 2006. ACM.
- [9] L. Bottaci. A genetic algorithm fitness function for mutation testing. In *SEMINAL 2001: International Workshop on Software Engineering using Metaheuristic Innovative Algorithms, a workshop at 23rd Int. Conference on Software Engineering*, pages 3–7, 2001.
- [10] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3), 1979.
- [11] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: adaptive random testing for object-oriented software. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 71–80, New York, NY, USA, 2008. ACM.

- [12] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software Practice and Experience*, 34(11):1025–1050, 2004.
- [13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
- [14] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [15] R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 549–552, New York, NY, USA, 2007. ACM.
- [16] G. Fraser and A. Arcuri. It is not the length that matters, it is how you control it. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST 2011)*, 2011, to appear.
- [17] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10*, pages 147–158, New York, NY, USA, 2010. ACM.
- [18] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, New York, NY, USA, 2005. ACM.
- [19] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.
- [20] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE 2008: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 297–306, 2008.
- [21] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions of Software Engineering*, To appear, 2010.
- [22] B. F. Jones, D. E. Eyres, and H.-H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41(2):98–107, 1998.
- [23] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? *TAIC-PART '09: Proceedings of Testing: Academic & Industrial Conference - Practice And Research Techniques*, 0:95–104, 2009.
- [24] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 267–276, Washington, DC, USA, 2005. IEEE Computer Society.
- [25] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *ASE '07: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 417–420, New York, NY, USA, 2007. ACM.
- [26] P. McMinn. Search-based software test data generation: a survey: Research articles. *Software Testing Verification Reliability*, 14(2):105–156, 2004.
- [27] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1:5–20, January 1992.
- [28] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Softw. Pract. Exper.*, 29:167–193, February 1999.
- [29] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5:99–118, April 1996.
- [30] A. Orso and T. Xie. BERT: BEhavioral Regression Testing. In *WODA 2008: Proceedings of the International Workshop on Dynamic Analysis*, pages 36–42, July 2008.
- [31] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 27–29, 2005.
- [32] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, New York, NY, USA, 2007. ACM.
- [33] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI 2005: Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2005.
- [34] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 69–80, New York, NY, USA, 2009. ACM.
- [35] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for Java. In *ESEC/FSE '09: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 297–298, New York, NY, USA, 2009. ACM.
- [36] D. Schuler and A. Zeller. (Un-)Covering equivalent mutants. In *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation*, 2010. To appear.
- [37] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, New York, NY, USA, 2005. ACM.
- [38] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th international conference on Software engineering, ICSE '08*, pages 351–360, New York, NY, USA, 2008. ACM.
- [39] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *ASE 2008: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410, 2008.
- [40] N. Tillmann and J. N. de Halleux. Pex — white box test generation for .NET. In *TAP 2008: International Conference on Tests And Proofs*, volume 4966 of *LNCS*, pages 134 – 253. Springer, 2008.
- [41] P. Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128, New York, NY, USA, 2004. ACM.
- [42] A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
- [43] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pages 1053–1060, New York, NY, USA, 2005. ACM.
- [44] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *EDCC 2005: Proceedings of the 5th European Dependable Computing Conference*, volume 3463 of *LNCS*, pages 281–292. Springer, 2005.
- [45] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP 2006: Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 380–403, July 2006.
- [46] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, Edinburgh, UK, April 2005.
- [47] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, September 2010.