Mutation-driven Generation of Unit Tests and Oracles

Gordon Fraser
Saarland University – Computer Science
Saarbrücken, Germany
fraser@cs.uni-saarland.de

Andreas Zeller
Saarland University – Computer Science
Saarbrücken, Germany
zeller@cs.uni-saarland.de

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 two open source libraries, our μTEST prototype generates test suites that find significantly more seeded defects than the original manually written test suites.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Algorithms, Experimentation

Keywords

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.

```
public class LocalDate {
  // The local milliseconds from 1970-01-01T00:00:00
  private long iLocalMillis;
  ...
  // Construct a LocalDate instance
  public LocalDate(Object instant, Chronology c) {
    ... // convert instant to array values
    ... // get iChronology based on c and instant
    iLocalMillis = iChronology.getDateTimeMillis(
        values[0], values[1], values[2], 0); ⇐ Change 0 to 1
  }
}
```

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

```
LocalDate var0 = new org.joda.time.LocalDate()
DateTime var1 = var0.toDateTimeAtCurrentTime()
LocalDate var2 = new org.joda.time.LocalDate(var1)
assertTrue(var2.equals(var0));
```

Figure 2: Test generated by μTEST. The call in Line 3 triggers the mutation; the final assertion detects it.

In this paper, we present μTEST, 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 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

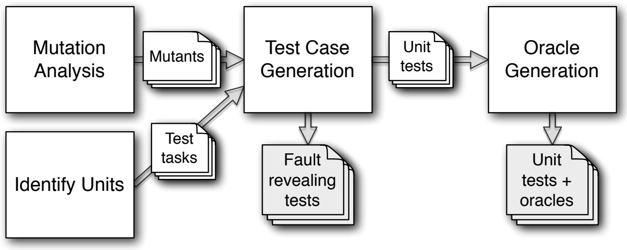


Figure 3: The μTEST process: Mutation analysis, unit partitioning, test generation, oracle generation.

last argument of `getDateTimeMillis` from `0` to `1`, thus increasing the value of `iLocalMillis` by 1, this seeded defect is not caught.

μTEST 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 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, μTEST 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 unit test case generation: μTEST uses a genetic algorithm to breed method/constructor call sequences that are effective in detecting mutants.

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.

Impact-driven test case generation: To minimize assessment effort, μTEST 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 applying a multi-objective approach that penalizes long sequences during test case generation, and then we remove all irrelevant statements in the final test case.

Figure 3 shows the overall μTEST 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 μTEST as an extension to the Javalanche [29] mutation system (Section 5). We demonstrate its effectiveness by applying it to two open source libraries that have a reputation of being extensively well-tested. The results show that our approach can be used to extend (and theoretically, replace) the manually crafted unit tests (Section 6). Section 7 closes with conclusions and future work.

2. BACKGROUND

2.1 Mutation Analysis

Mutation analysis was introduced in the 1970s [11] 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 [17] concisely summarizes all applications and optimizations that have been proposed over the years.

Javalanche [29] 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 [37]), 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 [14] and CUTE [31], and is also applied in Microsoft's parametrized unit testing tool PEX [33] or the object-oriented testing framework Symstra [39].

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, 22] as well as object-oriented container classes [4, 34, 35]. A promising avenue seems to be the combination of evolutionary methods with dynamic symbolic execution (e.g., [16, 19]), 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 [26], JCrasher [10], AutoTest [9], 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 [12] 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.

Jones et al. [18] proposed the use of a genetic algorithm to find mutants in branch predicates, and Bottaci [7] proposed a fitness function for genetic algorithms based on the constraints defined by DeMillo and Offutt [12]; recently this has been used for experiments using genetic algorithms and ant colony optimization to derive test data that kills mutants [5].

Differential test case generation (e.g., [13], [24], [32]) 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

In the context of regression testing, automated synthesis of assertions is a natural extension of test case generation. Randoop [26] allows annotation of the source code to identify observer methods to be used for assertion generation. Orstra [38] 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 [6]. 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 [25] can generate assertions based on a model learned from assumed correct executions; in contrast, μTEST does not require any existing executions to start with.

Evans and Savoia [13] generate assertions from runs of two different versions of a software system and DiffGen [32] extends the Orstra approach to generate assertions from runs on two different program versions. This is similar to our μTEST 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 a genetic approach in line with previous work on testing classes [4,34]: 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

Algorithm 1 Random generation used for the initial population.

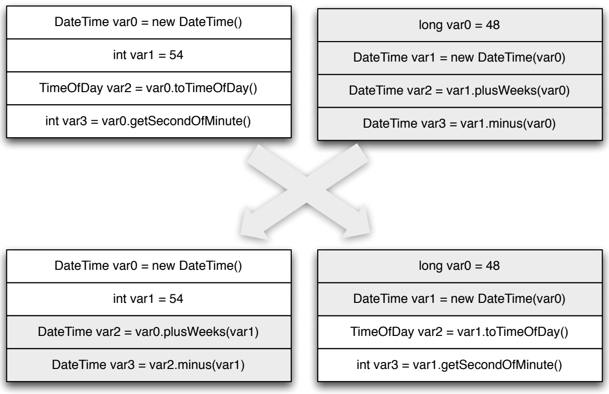
```
Require: $C$: Set of all methods and constructors directly/indirectly calling mutation
  $t \leftarrow \langle\rangle$
  $s \leftarrow$ random generator for unit under test
  for callee and all parameters of $s$ do
    GenObject(class of parameter, {}, $t$)
  end for
  $t \leftarrow t.s$
  while $|t| <$ 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 $t$
```

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 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.



DateTime var0 = new DateTime()
int var1 = 54
TimeOfDay var2 = var0.toTimeOfDay()
int var3 = var0.getSecondOfMinute()
long var0 = 48
DateTime var1 = new DateTime(var0)
DateTime var2 = var1.plusWeeks(var0)
DateTime var3 = var1.minus(var0)
DateTime var0 = new DateTime()
int var1 = 54
DateTime var2 = var0.plusWeeks(var1)
DateTime var3 = var2.minus(var1)
long var0 = 48
DateTime var1 = new DateTime(var0)
TimeOfDay var2 = var1.toTimeOfDay()
int var3 = var1.getSecondOfMinute()

Figure 4: Crossover between two 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 another statement, unless it can be replaced with another variable.

Insert an object: Create a new object of a type already used in the test case.

Insert a method call: Add a random method call for one of the objects of 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:* Replace a method call with a different one with identical return type.
- *Change constructor:* Replace constructor call with a different constructor of the same class.
- *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.

Which mutations are applicable for a given chromosome at a certain point is very dependent on the test case represented by the chromosome and the objects and calls it consists of. A chromosome is mutated by identifying all possible mutations and then randomly choosing mutations from this set until a mutation succeeds.

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:

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

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

We set the value of "Have callee" to 1 if the mutated method is static or the callee exists, but it would also be feasible to put more weight on achieving a callee object than on the parameter objects, because the object will also be useful for further method calls.

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 and branch distance measurement applied in search-based test data generation [22]. In addition, one can use the necessity condition [12] to estimate the distance to an execution of the mutation that infects the state (necessity distance [7]). 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.

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

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 [12]. 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 [28]: 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 [30] 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. As short test cases are preferable from a user perspectice we have a multi-objective optimization that aims to maximize impact and minimize test case length, and we have to combine impact and length in the fitness function:

$$I_m(t) = \frac{c \times |C| + r \times |A|}{1 + |t|} \quad (4)$$

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:

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

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 is not sufficient to stop once a mutant has resulted in an observable difference as we are also optimizing with respect to the test case length. Consequently, the genetic algorithm iterates 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 of method calls:

```
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 [27]. As an example:

```
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.

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 [8]. 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 μTEST approach as an extension to the Javalanche [29] 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, μTEST 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 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.

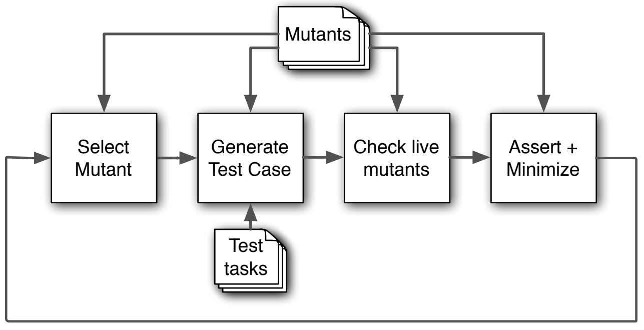


Figure 5: The process of generating test cases for one unit.

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 [9]. 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.

The information about test methods and mutations is output to files such that the tester can modify them at will, for example to exclude certain methods or add external helper functions.

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 [27] 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. μTEST selects a target mutant and generates a test case for it (see Figure 5). To measure the fitness of individuals, μTEST 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 fitness function penalizes long test cases and they are therefore usually short to begin with. Alternatively, one could for example apply delta-debugging [20] or a combination of slicing and delta-debugging [21] 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 un-loading 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 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 μTEST to two open source libraries. Test case generation only requires Java bytecode, and we used the prototype "out of the box" as far as possible, i.e., without writing test helper functions and adding as few as possible constraints. Minor adaptations were only necessary to avoid calling methods with nondeterministic results like random number generators; these methods are easily detectable as a test case calling such a method will achieve an unrealistically high mutation score, and assertions are not guaranteed to hold on the original program either. For both libraries we generated complete test suites with μTEST, and compared them with the already existing unit tests. The population size of the genetic algorithm was set to 100, with an initial test case length of 20 statements, and evolution was performed for 50 generations for each run.

6.1 Case Study Subjects

Joda-Time

As first subject for experimentation we chose the Java library Joda-Time[1], which provides a quality replacement for the Java date and time classes. The design allows for multiple calendar systems, while still providing a simple API and an extensive set of utility functions. 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.

Joda-Time has a number of aspects that are problematic with respect to testability:

Access rights are problematic at several points, granting only default (package) access. Such classes can be tested by unit tests residing in the same package—the test generation tool, however, will not be part of this package. To overcome this problem, we modify the bytecode during class loading, changing all default access modifiers to public access modifiers. When writing out the JUnit test cases at the end of the test case generation process they can be put into the appropriate package, giving them proper access to classes with package access restrictions.

Polymorphism is applied in a way that is problematic with respect to testability. For example, many classes of the "field" sub-package have private constructors and can only be generated via static getter methods. Many such getter methods, for example `public static DateTimeField getInstance(DateTimeField field)` of the class `StrictDateTimeField`, do not declare the actual return type (`StrictDateTimeField`) but the type of the superclass. In some cases this means there is no direct way to generate classes of this type, as dynamic type information is not available when test cases are constructed. We overcome this problem by adding casts as a new statement type, and restricting casts to superclasses only in those cases where no constructors are available.

Unspecified assumptions on the input parameters are quite common in Joda-Time: For example, in the constructor `public BaseDateTime(int year, int monthOfYear, ...)` the value of `monthOfYear` has to be in the range 0–11, or else an undeclared `IllegalArgumentException` is raised. Similarly, *polymorphic parameters* of type `Object`, for example in `public BaseDateTime(Object instant, ...)`, are quite common. The test case generation approach will try to put any object as parameter `instant`, as there are no restrictions on the type (every class is a subclass of `Object`). Unsupported classes, however, result in an undeclared exception.
We have overcome this problem with two workarounds: The first one is to allow specification of constraints on the parameter classes and value ranges, which is essentially a possibility to support contracts. The second option we used is to list exceptions that should not be reported as faults but ignored, with a penalty given to the fitness of individuals that raise such exceptions.

Determinism is not guaranteed, as a date and time library often accesses the current system time. This would render any assertions based on these values unusable, but fortunately Joda-Time offers a possibility to overcome this problem: There is a global `SystemMillisProvider` which returns the current time in milliseconds, and this can be configured in a test setup method to return a constant value.

[1]http://joda-time.sourceforge.net/

Table 1: Case study subjects (testable/total)

Case Study	Classes	Unit Tests	Mutants
Joda-Time	123/220	3,493	14,778/23,145
Commons-Math	220/413	1,739	25,226/43,273

Table 2: Manual vs. generated test cases

		Average per test case		
Case Study	Test cases	Lines	Assert.	Mut.
Joda-Time (man.)	3,493	4.89	4.55	3.14
Joda-Time (μTEST)	1,268	8.48	3.30	9.67
Commons-Math (man.)	1,739	7.97	3.41	5.98
Commons-Math (μTEST)	2,706	13.29	2.93	5.46

String parameters are currently generated by μTEST only by calling methods that return `String` objects, which reduces the success ratio for methods requiring such parameters.

Commons-Math

The second case study subject is Commons-Math[2], a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language. Again there is a certain degree of randomness in the library (for example in the sub-package that implements a genetic algorithm) which we excluded. Another problem with Commons-Math is the 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`.

Common-Maths contains classes for matrix and vector operations. Often, methods of these classes take arrays as parameters, e.g. in the `FieldMatrix` class:

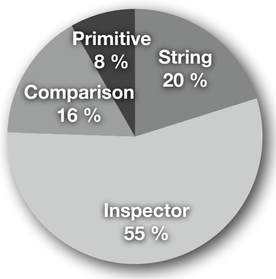
```
void copySubMatrix(int[] selectedRows, int[]
    selectedColumns, T[][] destination)
```

We currently generate arrays of primitive and non-primitive types (and thus also character strings) only via return values of other methods and fields. For Commons-Math this means that a certain share of the mutants (for example those in methods using FieldMatrix objects) currently cannot be tested properly.

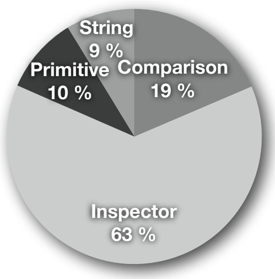
6.2 Results

Table 1 lists the numbers of testable classes and mutants for the case study subjects, and Table 2 gives more detailed statistics on the test suites provided by the case study subjects as well as those generated by μTEST: The average number of non-commenting statements excluding assertions, the average number of assertions per test, and the average number of mutants (out of the set of testable units) killed per test. The default behavior of μTEST is to only return a test case if it results in a non-zero impact for the considered

[2]http://commons.apache.org/math/



(a) Joda-Time



(b) Commons-Math

Figure 6: Average distribution of assertion types.

Table 3: Manual vs. generated tests: Mutation scores (ratio of detected mutations)

	Case Study	Manual	μTEST	Σ
Total	Joda-Time	74.26%	82.95%	90.34%
	Commons-Math	41.25%	58.61%	61.15%
Avg./Unit	Joda-Time	70.36%	80.36%	88.35%
	Commons-Math	44.50%	65.87%	67.72%

mutant, i.e., the test cases in Table 2 do not include cases where the mutation was executed without impact.

Figure 6 illustrates the distribution of the different types of assertions. Finally, Table 3 summarizes the ratio of killed mutants (the *mutation score*) for the existing test suites, the automatically generated test suites, and the combination of the two test suites, calculated from the total number of mutants for a project and on average per unit. A mutant is counted as killed if there is an assertion that detects it or it leads to an unexpected exception.

6.3 Discussion

Table 3 shows that in terms of the mutation score the results are better than the manually generated test suites. As both Joda-Time and Commons-Math are quite extensively tested (cf. Table 1), this is a very good result: The total number of mutants killed on the considered units is larger for both Joda-Time and Commons-Math, and also when viewed on average per unit μTEST can improve over both test suites. The difference is statistically significant according to a two-tailed Mann-Whitney test ($\alpha = 0.05$).

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

Keep in mind, though, as shown in Table 1, that not all units are testable with μTEST as discussed in Sections 5.2 and 6.1; Table 3 only counts mutants of testable units. In both libraries, μTEST increases the mutation score when combining the manually and automatically generated unit tests, which shows that the manual tests find some mutants that μTEST misses.

Although μTEST applies minimization to keep the test cases as short as possible and the number of assertions small, automatically generated test case are mostly not as easy to understand as manually written test cases, which is not surprising as the search algorithm can combine arbitrary statements. In our future work, we shall leverage actual *usage patterns* [36] to make test cases more similar to client code.

Table 2 shows that on average automatically generated test cases are longer than manually generated test cases, despite the minimization. However, the manually generated test cases build on a significant code base of shared setup methods: Joda-Time has 3856 non-commenting source code statements of general test code, and Commons-Math has 6579 statements. This code is re-used in many test cases thus comparison with the automatically generated test cases is difficult.

In terms of numbers, the vast majority (98.4%) of all test cases have at least one assertion; Table 2 shows that the number of assertions typically is around 3, which due to the assertion minimization is slightly less than for manually generated test cases. As shown in Figure 6, inspector assertions are the most common assertions, followed by comparison assertions. Surprisingly, primitive assertions only make a small share of all assertions. String assertions are more common in Joda-Time than in Commons-Math, as Joda-Time involves a number of string operations. Inspector assertions are preferable over other types of assertions, as they are a means to talk about the state of an object without revealing its internals. Comparing string representations, on the other hand, might reveal internal details and would thus result in assertions that are not proof to future change of the class.

Although the achieved mutation score is already quite high, the search based approach offers potential for optimization. While the coverage based impact measurement nicely guides the search towards assertions, the construction of call sequences to reach and execute mutations has a very rough fitness landscape. This could, for example, be improved by applying a chaining approach to identify relevant method calls [23], and taking object states into account.

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 μTEST 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++.

6.4 Threats to Validity

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

- As our 353 classes investigated come from two projects only, 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), 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.

- 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). Potentially, the generated oracles might 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 μTEST 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 [15], 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. Finally, adding intelligent string handling will immediately increase the applicability to a wide range of additional libraries.

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, and Valentin Dallmeier, Andrzej Wasylkowski, and the anonymous reviewers for comments on earlier versions of this paper.

8. 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 and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[5] 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.

[6] 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.

[7] L. Bottaci. A genetic algorithm fitness function for mutation testing. In *SEMINAL 2001: International Workshop on Software Engineering using Metaheuristic Inovative Algorithms, a workshop at 23rd Int. Conference on Software Engineering*, pages 3–7, 2001.

[8] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3), 1979.

[9] 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.

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

[11] 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.

[12] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. Technical Report TR-09-06, CREST Centre, King's College London, London, UK, September 2009.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] P. McMinn. Search-based software test data generation: a survey: Research articles. *Software Testing Verification Reliability*, 14(2):105–156, 2004.

[23] P. McMinn and M. Holcombe. Evolutionary testing using an extended chaining approach. *Evolutionary Computing*, 14(1):41–64, 2006.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *ASE 2009: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, November 2009. To appear.

[37] 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 ot the 5th European Dependable Computing Conference*, volume 3463 of *LNCS*, pages 281–292. Springer, 2005.

[38] 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.

[39] 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.