Generating Parameterized Unit Tests

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

State-of-the art techniques for automated test generation focus on generating executions that cover program behavior. As they do not generate oracles, it is up to the developer to figure out what a test does and how to check the correctness of the observed behavior. In this paper, we present an approach to generate *parameterized unit tests*—unit tests containing symbolic pre- and postconditions characterizing test input and test result. Starting from concrete inputs and results, we use test generation and mutation to systematically generalize pre- and postconditions while simplifying the computation steps. Evaluated on five open source libraries, the generated parameterized unit tests are (a) more expressive, characterizing general rather than concrete behavior; (b) need fewer computation steps, making them easier to understand; and (c) achieve a higher coverage than regular unit tests.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions, Pre- and Postconditions*

General Terms

Algorithms, Experimentation

Keywords

Test case generation, unit testing, test oracles, assertions

1. INTRODUCTION

The past decade has seen tremendous progress in automated test case generation. It is now possible to efficiently create sequences of method calls that achieve high code coverage. Code coverage does not mean much, though, if neither the test nor the unit under test contain assertions that assess computation results. Unless one wants to check for run-time exceptions, the tester is thus left with the job of adding postconditions (test oracles) that verify the observed behavior. This is a difficult task, as to check a test's output requires understanding what a test actually does—indeed, machine generated tests are often not meaningful.

ISSTA '11, July 17-21, 2011, Toronto, ON, Canada


```
void concrete_test() {
  YearMonthDay var0 = new YearMonthDay();
  TimeOfDay var1 = new TimeOfDay(var0);
  CopticChronology var2 = (CopticChronology)org.joda.time.Chronology
      .getCopticUTC();
  FixedDateTimeZone var3 = (FixedDateTimeZone)var2.getZone();
  DateTime var4 = var0.toDateTime(var1);
  DateTime var5 = var4.withZone(var3);
}
```

Figure 1: Concrete unit test generated for covering the `DateTime.withZone()` method in Joda-Time. Four out of the six statements are just setup (is it important that we are using CopticChronology?); also note the absence of oracles.

```
void parameterized_test(TimeOfDay input1, DateTimeZone input2,
      YearMonthDay input3) {
  assume(input2 != null);
  assume(input2.isFixed());
  assume(input3 != null);
  assume(input3.size() == 3);

  DateTime var0 = input3.toDateTime(input1);
  DateTime var1 = var0.withZone(input2);

  assertFalse(var0.equals(var1));
  assertNotNull(var1);
}
```

Figure 2: Parameterized test case generated from the test case in Figure 1 by systematically deriving new inputs and checking effects on seeded defects. Note the usage of pre- and postconditions, as well as the drastically reduced computation.

As an illustrating example, consider the code in Figure 1, a generated test for the `DateTime` class in *Joda-Time*, a replacement for the Java date and time packages. This test can be executed automatically; if any of the invoked methods crash, it will uncover the problem. As humans, though, we do not know what it does, what it expects, and which of its details are actually relevant. For instance, it instantiates a `CopticChronology` object—but not for testing the coptic calendar system. It needs some arbitrary `Chronology` object to use as a target for `getZone()`, which is needed to compute a `TimeZone` object to—finally!—cover the `withZone()` method, which was its purpose all along.

To aid the tester, one can use mutation testing to identify possible oracles [8]. But even so, the generated tests remain hard to understand because of the *amount of information specific for a single run.* This amount of detail not only hinders comprehension, it can also lead to test oracles that are overfitted to a particular run, thus making them brittle to future code changes.

To overcome these problems, we present a novel approach that takes a machine-generated method sequence like that shown in Figure 1, and explores both, possible *preconditions* and the relevant *postconditions* they imply. The resulting parameterized unit test contains only a small set of relevant pre- and postconditions, and only statements related to the actual test, as shown in Figure 2.

How does this work? Our approach builds on two key ideas. First, we separate *test code* from *test input*. This allows us to drop a large share of generated code that simply prepares (often random) input, and replace it by symbolic parameters. This significantly reduces the size of test cases and makes them more general at the same time.

The second key idea of our approach is that important behavior can be distinguished from irrelevant behavior by *variation*: By seeding defects (mutations), we can change some of the postconditions. Our assumption is that only those parts of the postcondition that are affected by the seeded defects are relevant; those that are not affected are irrelevant. In other words, the more errors a postcondition catches, the more relevant it is.

The same idea can be applied to the input: If we change the input, this might also change the postcondition. If a change in the input has no effects, then the particular aspect of the input that has changed seems to be irrelevant for the test. Furthermore, we can filter postconditions that are more sensitive to changes in the input, and determine those that are more robust. Similar techniques are used to generalize and select preconditions. Eventually, we combine the generalized pre- and postconditions with the test case into a *parameterized unit tests*, containing all information in a single package.

In detail, the contributions of this paper are:

Test parameterization: We convert concrete method sequences into parameterized test cases, reducing the number of statements the developer has to analyze.

Postconditions: By mutating the tested class, we identify the relevant aspects of the postconditions, and suggest oracles that are effective at finding defects.

Preconditions: By mutating the test inputs, we identify the relevant aspects of the preconditions, and filter out overly specific postconditions.

Iterative refinement: We use a search-based approach to iteratively derive new test inputs that aim at removing further preconditions, thus simplifying the test case.

Figure 3 gives a high-level overview of our approach: Based only on existing bytecode, we start with an automatically generated concrete method sequence (Section 2). Such a concrete method sequence has a very precise but implicit precondition; this precondition is encoded in the input objects and the setup performed on the unit under test (UUT). Similarly, the postcondition can be interpreted as the observable state after the test execution. We make these conditions explicit by determining all the conditions that hold for the given states (Section 3). For example, we compare all objects with each other, query all observer methods, and so on. The resulting conditions (d) overspecify the test case, and we therefore try to get rid of as many conditions as possible. For this, we iteratively generate new tests, and execute them on the original program and versions with seeded defects, thus effectively filtering irrelevant preconditions and postconditions (Section 4). At the end of the process, we get a parameterized unit test that only contains the test statements, the relevant preconditions on the inputs, and an effective test oracle.

2. BACKGROUND

2.1 Automated Test Generation

The importance and the complexity of software testing have 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 consider classical white-box testing, where test cases are derived from the program, and there are no assumptions about the existence of a formal specification.

A simple but effective approach to generate tests from programs is random testing. A main advantage of this approach is that it scales without problems, and experience shows that random testing can achieve relatively good coverage. Random testing tools such as Randoop [16], JCrasher [5], AutoTest [4], or RUTE-J [1] usually assume the existence of an automated oracle in terms of a specification or simply in terms of program crashes, as random tests are seldom meaningful but often long and complex, and trying to understand them in order to derive oracles can be a daunting task.

Often, coverage criteria are used to guide test generation and to reduce the number of tests that are generated. Such criteria can be defined over structural properties of the control-flow, data-flow properties, or fault-based properties such as used in mutation testing. A coverage criterion usually represents a set of distinct goals, and test generation techniques are invoked for all such goals that the current test suite does not yet cover. There are two main approaches to automate this task:

Constraint solvers allow efficient generation of test data by solving path constraints generated with symbolic execution (e.g., PathCrawler [25]). Dynamic symbolic execution is an extension that overcomes a number of problems by combining concrete executions with symbolic execution. This idea has been implemented in tools like DART [9] and CUTE [19], and is also applied in Microsoft's parameterized unit testing tool PEX [22] or the Symstra testing framework [27].

Meta-heuristic search techniques have been used as an alternative to constraint-based approaches [14]. They can also be applied to stateful programs [14] as well as object-oriented container classes [2, 23, 24]. As both constraint-based testing as well as search-based testing have their specific drawbacks, a promising avenue seems to be the combination of evolutionary methods with dynamic symbolic execution (e.g., [10, 11]).

2.2 Automated Oracle Generation

The test generation approaches listed above traditionally only generate *test data*, i.e., they produce inputs that execute a given coverage goal, but unless there is an automated oracle it is the developer's task to come up with efficient oracles. There is a growing awareness of the significance of this problem, and this has resulted in several related approaches.

In the context of regression testing, the objective is to check future program versions against the current state of the program, and it can therefore be used as a source for oracle generation. For example, Randoop [16] allows annotation of the source code to identify observer methods to be used for assertion generation. Return values and object states are used for assertion generation in tools like Orstra [26], and have also been adopted in commercial tools such as Agitar Agitator [3]. Evans and Savoia [7] generate assertions from runs of two different versions of a software system and DiffGen [20] extends the Orstra approach to generate assertions from

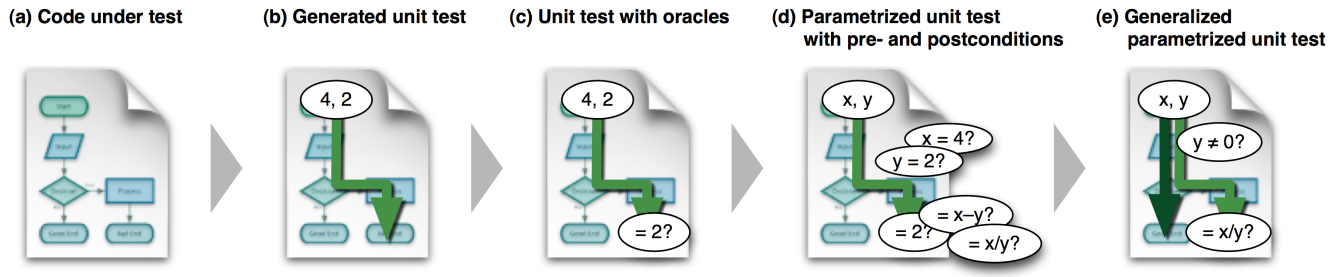

Figure 3: Our approach in a nutshell. From source code (a), we automatically generate concrete test cases (b) for any given test criterion. We derive test oracles from the concrete test results, checking the results (c). Concrete inputs and outputs can also be expressed as pre- and postconditions on general, symbolic inputs, coming as parameterized test cases (d). Using test generation and mutation, we iteratively drop irrelevant pre- and postconditions, ending up with a simpler, yet more expressive representation of the original concrete sequence as a parameterized unit test (e).

runs on two different program versions. Because these approaches are based in a regression testing scenario, understandability is not a requirement, and they do not serve to identify which of these assertions are actually useful.

Eclat [15] takes the oracle generation a step further by first learning a model from assumed correct executions, and then identifying test inputs that are different to this learned operational model. The operational model can be turned into a set of assertions that are effective at finding deviations of the behavior observed in the assumed correct executions. Eclat is based on Daikon [6], which is related to the approach presented in this paper: Daikon infers pre- and postconditions by observing executions, while the approach presented in this paper generates tests to explore and systematically eliminate irrelevant specifications.

To identify a small set of oracles that is effective at detecting faults, we used mutation analysis in our μTEST tool [8]. The central idea is that an assertion is only useful as a test oracle if it has the potential to reveal a fault. By mutating the source code, μTEST selects those assertions for a given test case that are sufficient to detect all mutations, thus also detecting most real faults. As the assertions are automatically generated, they reflect the actual behavior of the code and not necessarily the desired behavior. The developer therefore has to confirm the validity of a suggested assertion. In this paper we build on the μTEST tool; μTEST selects effective postconditions, and the main contributions of this paper are the parameterization and generalization steps.

2.3 Test Generalization and Test Factoring

A traditional unit test for object oriented software consists of setup code that prepares the test input, then executes some test code on the unit under test, and finally assesses the observed behavior. The recent success of parameterized unit testing is based on the ability to cover diverse behavior with a single case: The same test code executed with different inputs can show different behavior; tools like PEX [22] can produce these inputs automatically.

Retrofitting of unit tests [21] is an approach where existing unit tests are converted to parameterized unit tests, by identifying inputs and converting them to parameters, and by generalizing assertions to oracles that hold independently of the input. This is very similar to the present approach, but classical retrofitting of unit tests [21] is a manual process and assumes existing pre- and postconditions.

An additional advantage of parameterized unit tests over concrete unit tests is that they are easier to understand: There is less test code, and this code is also independent of the concrete inputs. Test factoring describes a related technique where an existing test case is converted to improve aspects such as readability or execution speed. For example, unrelated objects can be replaced with mock objects [17], and minimization [12, 13] reduces the length of automatically generated test cases.

3. PARAMETERIZING UNIT TESTS

Automated test generation for object oriented software creates sequences of method calls [10, 23, 27], where calls to the UUT are mixed with other calls that are required to create complex input objects. This makes it more difficult to see what is really tested, and makes the task of writing or checking assertions even more difficult. In addition, different inputs might yield different results, checking different parts of the code. This insight has resulted in the creation of *parameterized* unit tests, where the actual test code is separated from its inputs. Concrete values for the inputs can then be automatically generated by tools such as PEX [22].

As an automatically generated method sequence usually has some particular objective such as reaching a branch in the code, we can separate test code from input code. This task is easier than the process of *retrofitting* existing unit tests [21]: Existing test cases are equipped with assertions, and the generalization step might violate these assertions, resulting in false positives. In our setting, however, we just assume a method sequence without any assertions to start with, and thus generalization is without such problems.

3.1 Test Generalization

To generalize a method sequence to a parameterized method sequence, we assume one dedicated class as UUT. All method calls on this UUT are part of the test code, and objects created by calls to the UUT are also part of the test; all remaining calls are considered to be inputs. In principle, the techniques described in this paper could also be used with different granularity of units, e.g., one could focus on individual methods.

A test case $t = \langle c_1, c_2, \ldots, c_n \rangle$ is a sequence of n calls. The length of the test case is the number of calls n. A call can be a call to a constructor, a method, or an assignment of a primitive value or object member to a test object.

Each call represents a value $value(c)$, which is the return value for a method, the object constructed in case of constructor calls, or the value assigned in case of primitive or field assignments. Furthermore, each call may depend on a number of other values. These are method and constructor parameters as well as the source object of a method call or a field access. Let $params(c)$ be the set of referenced values, then the following must hold:

$$\forall c_i \in t : \forall p \in params(c_i) : \exists c_j : j < i \wedge value(c_j) = p$$

A parameterized unit test (PUT) $P = (I, T, Pre, Post)$ consists of a set of inputs I, a sequence of test calls T, and sets of pre- and postconditions *Pre* and *Post*. The addition of input values I changes

Algorithm 1 Parameterize Test Case

Require: Call Sequence $M = \langle m_1, \ldots, m_n \rangle$
Require: Class under Test C
Ensure: Parameterized Unit Test $P = (I, T, \mathit{Pre}, \mathit{Post})$

```
1:  procedure PARAMETERIZE(M, C)
2:      G ← (V, E)
3:      S ← {}
4:      for all m ∈ M do
5:          v ← value(m)
6:          V ← V ∪ {v}
7:          for all v' ∈ params(m) do
8:              G ← G ∪ {(v', v)}
9:          end for
10:         if m is a call of C then
11:             S ← S ∪ {v}
12:             T ← T.m
13:         end if
14:     end for
15:     for all v ∈ S do
16:         for all (v', v) ∈ V do
17:             if v' ∉ S then
18:                 I ← I ∪ { New parameter with type of v}
19:                 p ← Backwards slice of v'
20:                 Pre ← Pre ∪ { Extract conditions for p }
21:             end if
22:         end for
23:     end for
24:     Post ← { Extract conditions for each value in T }
25:     return P
26: end procedure
```

the possible dependencies between calls:

$$\forall c_i \in t : \forall p \in \mathit{params}(c_i) : (\exists c_j : j < i \wedge \mathit{value}(c_j) = p) \vee p \in I$$

Algorithm 1 shows how a call sequence is converted to a parameterized unit test. First, we generate a graph in which there is a vertex for every value, and edges between values if the call producing a value has dependencies on other values. By separating values that are test code from those values that are setup code, one can easily determine inputs: For each test vertex there is one input for every incoming edge that does not come from another test vertex. The graph easily lets us derive method sequences to construct each of the parameters; the calls that are part of the test and not part of an input are added to T. If the same input value is used by different calls, then each of the uses results in a distinct input.

3.2 Extracting Pre- and Postconditions

Besides a set of statements and inputs, a PUT also consists of preconditions, i.e., assumptions on the inputs, and postconditions, i.e., test oracles that check the observed behavior during test execution. When converting a concrete unit test to a parameterized unit test, we initially have very specific but *implicit* preconditions, represented by the actual test inputs. Similarly, the test case has an implicit postcondition given by the state at the end of the test execution. To make these conditions explicit, we capture all possible conditions on the inputs and test objects.

We derive conditions by all accessible means of the public API, which means that the vocabulary for conditions and thus the precision of the approach depends on the richness of API. For example, things we can check on input values and test objects are:

- Null reference checks
- Values of primitive variables
- Values of public primitive fields
- Return values of calls to inspector methods (i.e., methods with no parameters, primitive return types, and no side-effects)
- Comparisons and equality checks between objects
- Comparisons with predefined values, such as the constant 0

Using the graph generated with Algorithm 1, we can extract a distinct method sequence for each of the input values of the original method sequence. To construct the initial precondition of a test case, we execute this method sequence for each of the parameters, and then determine the set of conditions that holds in the state at the end of the execution (Line 20 in Algorithm 1). For example, for each $i \in I$, we

- add $i = null$ if i is a null reference, else we add $i \neq null$ to *Pre*
- for each $i' \in I : i \neq i'$ where the type of i equals type of i' we add $i.equals(i')$ or its negation depending on the output
- for each inspector method m of i, we add a condition that $i.m() = x$, where x is the value of m observed after executing it on i.
- for each field f of i, we add a condition that $i.f = x$, where x is the value of m observed after executing it on i.

Similarly, the set of postconditions *Post* is determined by executing the concrete method sequence, and then extracting all conditions on values of test calls at the state at the end of the execution (Line 24 in Algorithm 1).

4. FINDING RELEVANT CONDITIONS

The PUT we generate from a method sequence with the procedure described above is semantically equivalent to the original method sequence only under the assumption that the public API allows a precise determination of the relevant states. In addition, the PUT has assertions (the postcondition) that detect all defects that the test can possibly detect, but it has two main problems:

- There are too many preconditions and postconditions, making it difficult to understand the test case. This, however, is necessary, as the postconditions need to be confirmed by a developer. We therefore need a way to determine the *effective* and *relevant* conditions.

- There are too many postconditions, checking every single observable aspect, such that future changes of the tested code are very likely to violate at least one of the assertions – even if the code change is correct. In other words, the assertions are not *robust* against code changes.

The problem is that this is in fact an *overspecification* of the test case; many conditions that can be derived this way are irrelevant and unnecessary for the task of detecting defects, and there are far too many such conditions to make it feasible for a developer to check every single of them.

4.1 Finding Effective Postconditions

To overcome this problem, we recently presented an approach [8] that identifies the important postconditions: The most useful postconditions are those that can identify *defects* in the code, and they can be determined by seeding artificial defects into the code. This idea is based on mutation analysis, which describes the observation that a test suite that can distinguish between a program and its simple mutants is generally good at detecting faults.

Algorithm 2 illustrates how a set of postconditions is reduced to the relevant subset: The test case t is executed against every single mutant, and a postcondition only qualifies as relevant oracle, if there exists at least one mutant for which the assertion fails.

Algorithm 2 Determine effective postconditions

Require: Call Sequence $M = \langle m_1, \ldots, m_n \rangle$
Require: Class under Test C
Require: Mutants of Class under Test $\mathcal{M}$
Require: Set of Postconditions *Post*
Ensure: Reduced Set of Postconditions *Post*′

```
1: procedure FINDEFFECTIVE(M, C, M, Post)
2:     Post' ← {}
3:     for all m ∈ M do
4:         S ← state after executing M on m
5:         for all p ∈ Post do
6:             if p evaluates to false in S then
7:                 Post' ← Post' ∪ {p}
8:             end if
9:         end for
10:    end for
11:    return Post'
12: end procedure
```

The set of conditions created with this reduction is not minimal: In fact, it can still be quite large, meaning that the original problem of having too many conditions is only reduced. In our previous work [8], we reduced the set of postconditions further to a minimal set in the sense that removing any of the postconditions will cause defects to be missed. There is, however, the possibility that the selected assertions are too specific to the current test input. As an extreme example, the hash-code of an object integrates all the information contained in the object, and an assertion on the hash-code is therefore likely to detect many mutants. Assertions on the hash-code, however, are clearly neither robust nor particularly meaningful.

4.2 Finding General Postconditions

We want our postconditions to be as *general* as possible – they should include as little concrete information as necessary to detect all defects that can be detected. For example, an assertion that compares equality of two instances of a class is more likely to hold true even after future code changes, than asserting that the return value of an inspector function has a specific value. At the same time, we do not want to sacrifice any of the fault detection capability of a given concrete method sequence.

To identify postconditions that are more general, a simple approach is to execute the PUT with different inputs: If the same assertion detects a mutation for different inputs, then it is more general than an assertion that only holds for a single concrete input. This insight be exploited in order to remove the overspecific postconditions: Algorithm 3 illustrates how a set of postconditions can be gradually reduced to retain only general postconditions that hold for more than one input. The PUT is repeatedly executed with new inputs, and an overspecific postcondition tied to a particular input is likely not in the intersection of the postconditions of two different inputs. However, if there is an alternative way to detect the mutation with a more general postcondition, then this general postcondition will remain in the intersection. This means that rather than selecting the most effective and potentially overfit postcondition, we can vary the inputs on the PUT to find a subset of assertions that is sufficient to detect all mutants, but stays as independent as possible from the concrete input values. Consequently, if the intersection of the postconditions of the new input and the current postcondition are sufficient to kill all mutants in Algorithm 3, then the new postcondition is this intersection.

The result of Algorithm 3 is a set of postconditions. Each of these assertions is general enough with respect to the inputs which were used to filter this list. A subset of this set of postconditions can detect the same defects, and we could therefore further apply a normal reduction procedure, as for example used in our previous work [8], where we traced for each assertion which mutation it kills, and then found a subset for each test case that is sufficient to detect all mutations that can be detected with this test case.

Algorithm 3 Determine robust postconditions

Require: PUT $P = (I, T, Pre, Post)$
Require: Class under Test C
Require: Mutants of Class under Test $\mathcal{M}$
Ensure: Reduced Set of Postconditions *Post*

```
1: procedure FINDROBUST(P, M, C)
2:     while limit not reached do
3:         I ← generate concrete inputs for I
4:         M ← concrete method sequence of I and T
5:         Post' ← FINDEFFECTIVE(M, C, M, Post)
6:         if Post' ∩ Post detects the same mutants as Post then
7:             Post ← Post' ∩ Post
8:         end if
9:     end while
10:    return Post
11: end procedure
```

An important insight is that generalizing postconditions by themselves is not enough: Although the postconditions have the *potential* to detect a number of defects, there is no guarantee that a given input will actually reveal a defect (false negative). Even worse, it might be possible to find inputs for which one of the assertions fails even in absence of a defect (false positive). The reason for this is that only a truly universal postcondition would hold for all inputs, whereas less general properties only hold under certain conditions – the test's precondition. It is therefore essential that when generalizing a postcondition we also need to derive an appropriate precondition. The original precondition *Pre* we extracted from the concrete unit test is likely to be very specific, which means that the number of possible inputs that satisfy this condition is very small, or possibly there exists exactly one input. We therefore need to enforce a PUT's precondition, but need to generalize it as well.

4.3 Finding Relevant Preconditions

A concrete input represents a number of specific preconditions and implies a number of specific postconditions, which have the potential to detect a number of defects. A key insight is that a precondition is only relevant if it is directly linked to the effective postconditions or mutants they detect, whereas we are not interested in preconditions that do not affect the postcondition, or affect only such assertions that are not effective at detecting defects.

To identify the relevant preconditions we therefore vary the inputs such that they satisfy different preconditions, and whenever the relevant behavior is identical for two different inputs we have found an irrelevant precondition. We define the behavior of two different inputs to a PUT to be equivalent only if the intersection of relevant assertions of the two sets of inputs kills all mutants that are also killed by the original test, as was done in Algorithm 3.

To systematically find out which preconditions are relevant, we need to consider each precondition on its own: If we can find two different inputs, where the considered precondition holds on one but not on the other while all other preconditions hold on both inputs, then we know that this precondition is irrelevant if the two inputs detect the same mutants with the same postconditions. If the two inputs would differ on more than one precondition, then it would not be clear which of the preconditions causes a possible change of the observed behavior.

Algorithm 4 Generate a Parameterized Unit Test

Require: Call Sequence $M = \langle m_1, \ldots, m_n \rangle$
Require: Class under Test C
Require: Mutants of Class under Test $\mathcal{M}$
Ensure: Parameterized Test Case $P = (I, T, Pre, Post)$

```
1: procedure GENERALIZE(M, C, M)
2:     P ← PARAMETERIZE(M, C)
3:     Pre' ← minimized observations on inputs
4:     Post ← observations on C after test execution
5:     Post ← FINDEFFECTIVE(M, C, M, Post)
6:     while Pre' is not empty do
7:         p ← remove one element from Pre'
8:         I ← generate input that satisfies ¬p ∧ ⋀ Pre' ∧ ⋀ Pre.
9:         if test generation succeeds then
10:            Post' ← Execute test with new input I
11:            M ← concrete method sequence of I and T
12:            Post' ← FINDEFFECTIVE(M, C, M, Post')
13:            if Post' ∩ Post detects all mutants then
14:                Post ← Post' ∩ Post
15:                Pre' ← Pre' ∪ conditions subsumed by p
16:            else
17:                Pre ← Pre ∪ {p}
18:            end if
19:        end if
20:    end while
21:    return P
22: end procedure
```

In order to apply this technique, a prerequisite is that the precondition is *minimal*. For example, if there are two inputs A and B, and we have the conditions $A = 1$, $B = 1$, and $A = B$, then negating any one of them while holding the other two true results in an unsolvable constraint system. We therefore minimize the set of preconditions before attempting to find inputs for negated conditions. In particular, we omit symmetric and transitive properties, as well as subsumed properties (e.g., $a < b$ subsumes $a \neq b$). However, if we can show that a precondition is irrelevant, then this does not automatically mean that subsumed or transitive conditions are also irrelevant. Therefore, we need to add all its subsumed conditions after a precondition has been shown to be irrelevant.

Algorithm 4 illustrates the algorithm to reduce the preconditions of a given PUT. Our algorithm starts with the precise precondition of the original method sequence, i.e., the entire set of conditions we can formulate. Now we iteratively try to negate one of the preconditions, and generate a new set of inputs that satisfies this negated precondition. For the new test input we calculate the output (i.e., the relevant part of the postcondition), and check the equivalence. If we can generate a test input for which a precondition of the previous inputs does not hold but the interesting behavior is still the same (i.e., the intersection of the relevant assertions kills the same mutants), then this precondition is irrelevant for the test case. If we cannot generate a test input that negates a precondition, then this precondition is likely an invariant, meaning that it applies to all possible inputs and is not relevant either.

4.4 Parameter Generation

To generate inputs for the PUTs may often mean generating complex objects. To make sure that the states of these objects are feasible, we generate them using a search-based approach in line with previous work on evolutionary testing of classes [23].

We use a genetic algorithm to derive inputs for a PUT. A chromosome of the genetic algorithm is a set of method sequences of the size of the number of parameters, i.e., one method sequence per parameter. The initial population is generated randomly by choosing from the set of constructors and methods that return an object of the desired type, and by recursively trying to create all necessary parameters. During evolution in the genetic algorithm, crossover exchanges parameter sequences between sets of parameter assignments. For mutation, each method sequence for a parameter is mutated with probability $1/n$, where n is the number of parameters of the PUT. Mutation of method sequences follows standard operators used in evolutionary testing of classes [8, 23]; for example, mutation adds, deletes, or changes method calls in the individual sequences. The last object of the parameter type is always returned as value for the input; if a method sequence has no such object, then the input is a null reference.

The fitness function aims to satisfy all in a set of preconditions, and is similar to the branch distance metric [14] commonly applied to predicates in source code. For example, to satisfy a condition $foo.bar() = 100$, the fitness function would use the absolute value of the difference of the actual return value of *foo.bar* and 100. The overall fitness is calculated as the sum of branch distances to have all preconditions evaluate to true, and if the search objective is to negate a single precondition, then the fitness additionally adds the branch distance to have this condition evaluate to false.

Table 1: Statistics on evaluation subjects.

Subject		Classes	LOC[5]	Mutants	Tests
Commons Codec	CC	18	2,640	2,998	285
Commons Math	CM	26	8,752	6,476	239
Java Collections	JC	23	7,250	4,990	346
JDOM	JD	16	2,286	820	199
Joda Time	JT	25	6,923	3,004	588
Σ		108	27,851	18,288	1,657

5. EVALUATION

To evaluate the presented approach, we have implemented a prototype that performs generation of parameterized unit tests for Java classes, and applied this prototype to a set of evaluation subjects.

5.1 Evaluation Subjects

As subjects for our evaluation we selected five open source libraries with very diverse functionality: *Commons Codec*[1] (CC) provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs. *Commons Math*[2] (CM) is a library of mathematics and statistics components. *Java Collections* (JC) is the set of collections in the Java standard library. *Joda Time*[3] (JT) provides replacement for the Java date and time classes. *JDOM*[4] (JD) provides a way to represent XML documents for easy and efficient reading, manipulation, and writing. For each of these libraries, we selected all concrete classes of the top level of the package structure for test generation; for Commons Math we selected the sub-packages complex, fraction, geometry, and linear because of the duration of the experiments. Table 1 summarizes statistics of the selected subjects in terms of classes, lines of code, mutants, and tests generated.

[1] http://commons.apache.org/codec/
[2] http://commons.apache.org/math/
[3] http://joda-time.sourceforge.net/
[4] http://www.jdom.org
[5] LOC stands for non-commenting lines of source code, calculated with CLOC (http://cloc.sourceforge.net/)

Table 2: Number of inputs generated and number of executions.

Subject	Iterations	Fail	Success	Inputs	Time (s)
Commons Codec	4.4	1.1	3.3	1.4	189.7
Commons Math	10.3	3.0	7.3	1.3	1,731.2
Java Collections	12.1	5.8	6.4	1.4	559.1
JDOM	10.5	3.1	7.3	1.3	366.3
Joda Time	9.1	4.8	4.3	1.5	644.6
⊘	9.3	3.6	5.7	1.4	689.2

5.2 Experimental Setup

Our prototype is built on our previous μTEST system [8] for automatic generation of mutation detecting unit tests and oracles. For a given class, the prototype first tries to generate a unit test (method sequences with oracles) for each mutant (skipping those that are covered by already generated tests); Table 1 lists the number of test cases generated for the chosen classes. Then, each of these unit tests is generalized to a parameterized unit test (PUT) using the techniques described in Section 3 and Section 4. The search for concrete parameter values uses the same data structures as μTEST, such that each parameter of a PUT is assigned the result of a method sequence.

We configured μTEST to use a steady state genetic algorithm with a population size of 100, an elitism size of 1 individual, and the maximum test case length was set to 60 statements. The initial test suite for a class was generated for a maximum of 1,000,000 executed statements per class. This budget was equally divided on all mutants, and for each mutant. 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 assumed that the mutant causes an infinite loop and ended the search for this particular mutant at this point. We used the same configuration to generate input sequences for parameters, but limited the search budget for one input to 500 generations. To reduce the risk of erroneously dropping a precondition because the search failed, we implemented the loop at Line 6 in Algorithm 4 such that each precondition was considered a second time at a later point, if the search failed. For condition generation we used all assertions described in Section 3.2 except for comparisons between inspector method return values and comparisons with predefined values.

5.3 Effort

The generalization of method sequences to PUTs is costly, as we experienced during our experiments. For each precondition, we have to generate a test case. Often, the conditions offer little guidance for a search based approach, letting the search exhaust. Many preconditions are not independent of each other, which means that often no solution for a single negated precondition exists. Furthermore, to determine the interesting behavior of a concrete input, the PUT with this input has to be executed on every single mutant of the class it tests.

To better understand the effort involved in the generalization, we take a closer look at the numbers of inputs that needed to be generated and the number of tests that need to be executed. The number of inputs generated might differ from the number of preconditions, as one precondition might subsume several others. If a condition is dropped, then we have to consider all conditions that were subsumed by this condition; if it is not dropped, then the subsumed conditions do not have to be analyzed.

Table 2 summarizes the average number of iterations for each of the subjects. One iteration equals one attempt of generating an input that satisfies all confirmed and remaining conditions as well as the negation of one chosen condition. Note, that in our implementation each precondition for which the search for an input satisfying the negated precondition failed is considered a second time, as described in Section 5.2. Such a search requires execution of an average of 500 (generations) times 100 (population) input sets consisting of one method sequence per parameter. Table 2 lists the average number of parameters per test for each of the subjects. If an input is found (Success in Table 2), then it is executed against each of the mutants; this again can be very costly, as the number of mutants can be high (see Table 1), and mutants can often lead to infinite loops, which means that they run up to the predefined timeout (5 seconds in our case). On average, it therefore took *12 minutes* to generalize a single test case on an Intel Xeon X5570 computer with 2.93GHz.

For the experimentation, we had the additional costs of generating the concrete method sequences in the first place, and of generating 1,000 valid inputs for each PUT. In sum, these costs required us to reduce the number of repetitions for each experiment.

Generating parameterized unit tests is costly, requiring several minutes per test case.

5.4 Examples

We have seen that much computation time goes into generating parameterized unit test. Does this effort pay off? Figures 4–7 show example method sequences and the resulting parameterized unit tests (PUTs) derived during our evaluation. In addition, the initial example in Figure 2 was also generated during this evaluation. In all cases, the parameterized unit tests generated are far more readable and meaningful compared to the original concrete tests.

Parameterized tests are more expressive than the original tests, characterizing general rather than concrete behavior.

```
void concrete_test {
  int var0 = 213;
  ArrayList var1 = new ArrayList();
  Integer var2 = new Integer(var0);
  boolean var3 = var1.add(var2);
  var1.trimToSize();
  int var5 = var1.indexOf(var2);
}
```

```
void parameterized_test(Integer input1, Integer input2) {
  assume(input1.equals(input2));

  ArrayList var0 = new ArrayList();
  boolean var1 = var0.add(input1);
  var0.trimToSize();
  int var3 = var0.indexOf(input2);

  assert(var3 == 0);
  assert(var1 == true);
  assert(var0.isEmpty() == false);
}
```

Figure 4: Example method sequence and resulting PUT for the `ArrayList` class in Java Collections: The object `input2` is at position 0 (`var3`) only if `input1` and `input2` are equal.

```
void concrete_test() {
  Caverphone var0 = new Caverphone();
  String var1 = "EL";
  String var2 = "ILLA";
  boolean var3 = var0.isCaverphoneEqual(var1, var2);
}
```

```
void parameterized_test(String input1, String input2) {
  assume(!input1.equals(input2));

  Caverphone var0 = new Caverphone();
  boolean var1 = var0.isCaverphoneEqual(input1, input2);

  assert(var1 == false);
}
```

Figure 5: Example method sequence and resulting PUT for the `Caverphone` class in Commons Codec: Non-equal strings stay different when encoded using the Caverphone algorithm.

```
void concrete_test() {
  Complex var0 = Complex.ZERO;
  double var1 = 0.39860618453401475;
  Complex var2 = var0.multiply(var1);
}
```

```
void parameterized_test(double input1) {
  Complex var0 = Complex.ZERO;
  Complex var1 = var0.multiply(input1);

  assert(var0.equals(var1));
}
```

Figure 6: Example method sequence and resulting PUT for the `Complex` class in Commons Math: The value of `input1` does not matter in a multiplication with 0, therefore there is no precondition.

5.5 Test Simplification

In the examples, we have seen that the number of statements in the generated parameterized unit tests is generally reduced. The first question we address in the evaluation is thus how much the generalization step reduces the number of statements that need to be analyzed by the developer. In a setting where test inputs are purely numerical, this step would not make a difference. However, our original motivation is that generated test cases for object oriented software tend to be cluttered with setup code, reducing the understandability.

Table 3 lists the mean values of the number of statements in the original concrete tests, compared to the number of statements that remain in the parameterized test; Figure 8 gives detailed statistics on these values. On average, the PUTs only contain 57% of the statements in their corresponding concrete method sequences—simply because 43% of the statements of the original method sequences generate inputs, which a PUT replaces with parameters. This reduction is statistically significant according to a Mann-Whitney U test with $p = 0.008$.

In our experiments, parameterized unit tests only retain 57% of the original statements in the concrete unit test.

5.6 Condition Generalization

The reduction of test statements clearly shows that the step of converting a concrete method sequence to a PUT reduces the number of statements that need to be analyzed and understood by a developer. This PUT and its postcondition, however, are still tied to the concrete inputs the original method sequence represents.

```
void concrete_test() {
  AttributeList var0 = new AttributeList((Element) null);
  UncheckedJDOMFactory var1 = new UncheckedJDOMFactory();
  String var2 = "namespace␣prefix␣on␣the␣element";
  Attribute var3 = var1.attribute(var2, var2, (Namespace) null);
  var0.uncheckedAddAttribute(var3);
}
```

```
void parameterized_test(Element input1, Attribute input2) {
  assume(input1 == null);
  assume(input2 != null);

  AttributeList var0 = new AttributeList(input1);
  var0.uncheckedAddAttribute(input2);

  assert(var0.isEmpty() == false);
}
```

Figure 7: Example method sequence and resulting PUT for the `AttributeList` class in JDOM: After adding a non-null element to the empty list, it is no longer empty.

Table 3: Average number of statements.

Subject	Concrete unit test	Parameterized unit test	Size
Commons Codec	3.9	2.2	56%
Commons Math	4.3	2.7	62%
Java Collections	5.6	3.2	57%
JDOM	5.0	2.7	55%
Joda Time	5.1	2.9	56%
∅	4.8	2.7	57%

The number of preconditions and postconditions can be high in theory, but their number is dependent on the API under test: If there are many inspector methods or public fields allowing many different queries on the state, then there will be many conditions. If the API hides most information and only offers few possibilities to inspect the state, then the number of conditions can be much smaller.

Table 4 summarizes the average statistics on the number of preconditions before and after the optimization. The average number of preconditions before the optimization is below 10. As can be seen from Table 4, condition generalization significantly reduces the number of preconditions. This result is again statistically significant according to a Mann Whitney U test with a p-value of 0.008.

Our approach for finding relevant preconditions eliminates 88% of the preconditions as irrelevant.

Table 5 shows similar results for the postconditions. The number of postconditions is generally smaller than the number of preconditions, as the really irrelevant ones are already filtered out by the mutation analysis step; only fault detecting postconditions are listed here. Any postcondition that is removed from the original set was specifically tied to an input, and represents generalization. The reduction is again statistically significant with a p-value of 0.008.

Our approach for finding general postconditions eliminates 46% of the postconditions as overspecified.

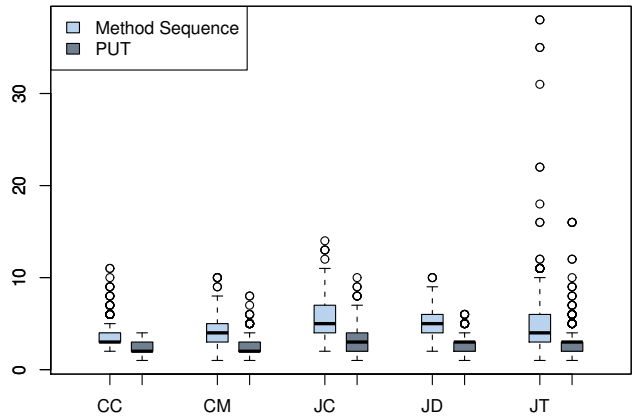

Figure 8: Numbers of statements in concrete method sequences compared to the number of statements in the PUTs.

Table 4: Average number of preconditions.

Subject	Original	Relevant	Size
Commons Codec	5.1	1.1	22%
Commons Math	6.8	0.7	10%
Java Collections	13.8	1.6	12%
JDOM	9.2	1.3	14%
Joda Time	8.1	0.5	6%
⊘	8.6	1.0	12%

5.7 Accuracy

Generalizing concrete runs to abstractions may produce *approximations:* While the original concrete input is always correctly handled by construction, generalizing it to parameterized abstract values may result in false positives or false negatives. This happens if the vocabulary of usable conditions is not sufficient to eliminate invalid inputs, for instance. Similarly, if the input generation fails for a particular negated precondition, then it is not known whether this precondition can really be dropped or not. (To only present the preconditions to the user that are known to affect the result, we dropped such preconditions.)

To analyze the extent of false positives and negatives for new inputs, we generated 1,000 different inputs satisfying the minimized precondition using different random seeds, and then executed the PUT with these inputs. If one of the postconditions fails on an input when run on the original class, then this is a false positive. If none of the postconditions fails on an input when run on all the mutants that should be detected, then this qualifies as a false negative. Table 6 summarizes the number of false positives and negatives, and Figure 9 shows detailed statistics on the accuracy.

For new inputs, the generated parameterized unit tests have a false negative rate of 19.6% (i.e., failing to detect faults).

The rate of false negatives is based on our strict requirement to detect *all* mutants that the original method sequence detected. This requirement could be weakened, for example by requiring that each postcondition detects at least one mutant. If we only count as false negatives when an input leads to detection of *less* mutants than the original method sequence, then the false negative rate would drop to 4.2%.

Table 5: Average number of relevant postconditions.

Subject	Fault detecting	Generalized	Size
Commons Codec	4.4	2.1	53%
Commons Math	4.2	2.7	64%
Java Collections	5.7	2.9	51%
JDOM	5.2	2.7	52%
Joda Time	5.4	3.0	56%
⊘	5.0	2.7	54%

Table 6: False positives and false negatives, evaluated with 1,000 different inputs per PUT.

Subject	False Negatives	False Positives
Commons Codec	23.5%	6.1%
Commons Math	26.0%	10.2%
Java Collections	26.8%	12.7%
JDOM	16.2%	5.9%
Joda Time	5.6%	6.4%
⊘	19.6%	8.3%

For new inputs, the parameterized unit tests have a false positive rate of 8.3% (i.e., failing although no fault is present).

Given that parameterized unit tests are far more general than the concrete original tests, the low false positives and false negative rates are a small price to pay for the added value—not to speak of readability or maintainability. On top, our setting used for *evaluating* accuracy can also be used to *increase* accuracy—simply by retaining those conditions that show no false positives or negatives, and refining the others. This way, as the number of generated inputs increases, so will accuracy improve further.

5.8 Coverage Increase

A test case with one concrete input will always follow the same execution path – its coverage is fixed. By generalizing the preconditions, other allowed test inputs might lead to different execution paths, resulting in different coverage. However, as we require generalization to preserve the fault detection ability, the execution paths may only vary as long as they do not affect the detected

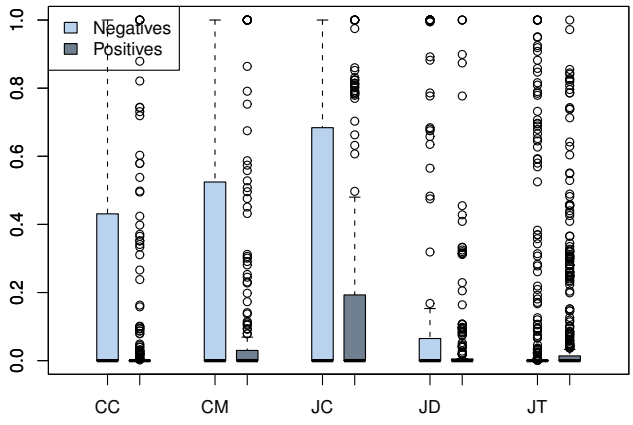

Figure 9: Rate of false positives and false negatives of the generalized PUTs.

Table 7: Average number of covered branches and increase in branch coverage.

Subject	Concrete unit test	Parameterized unit test	Coverage Increase
Commons Codec	23.6	51.4	118.4%
Commons Math	10.0	12.7	26.5%
Java Collections	13.0	18.9	44.8%
JDOM	7.7	10.3	34.5%
Joda Time	5.2	8.7	68.0%
⊘	11.0	19.0	72.6%

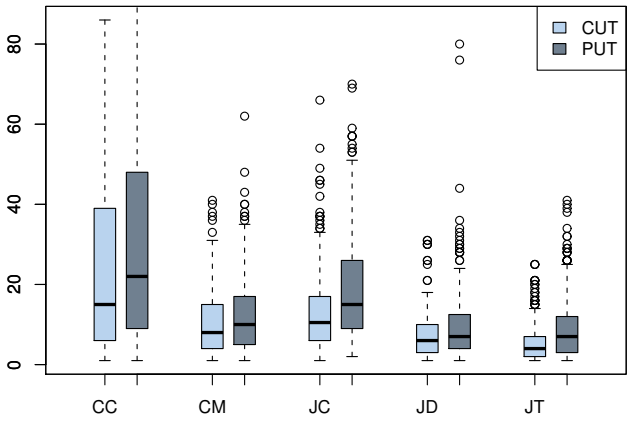

Figure 10: Statistics on the branch coverage of concrete method sequences compared to the generalized PUTs.

faults (i.e., the identical mutants need to be executed and their state changes need to be propagated). We therefore expect a small but measurable increase in the coverage. The quantify this effect, we measured the coverage of the original method sequence, and also measured the compound coverage achieved by all valid inputs (i.e., detecting the same defects with the same postconditions) generated during the test case generation. Figure 10 illustrates this increase using branch coverage; we see that the generalized parameterized unit tests are far more effective than the original concrete tests.

On average, generated parameterized unit tests cover 72.6% more branches as the original concrete unit test.

There are two ways in which the coverage could be further increased: First, by focusing a single PUT on a single mutation the variability in the taken execution paths would increase, thus allowing higher coverage. Second, each time we detect a relevant precondition, we could in theory create a new PUT that contains the negation of the currently considered precondition. This would increase the number of test cases, but it would also increase the coverage achieved with the same set of test statements.

6. THREATS TO VALIDITY

Threats to *construct validity* are on how the improvement of the test parameterization is defined. We measured improvement of the understandability in terms of the reduction of statements a user has to analyze as well as the reduction of the conditions that need to be analyzed. A reduction of elements that need to be considered is an intuitive indication of simplification, but understandability cannot be quantified – ideally, one would need human experiments for this.

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of defects in our testing framework, it has been carefully tested. As randomized algorithms are affected by chance, we used statistical procedures to evaluate the result of the experiments. Our findings might be biased by the concrete method sequences and input sequences derived by our own tool μTEST; however, μTEST implements state-of-the-art techniques to generate method sequences [23]. The approach is also dependent on the number and type of mutants seeded; we used Javalanche [18] to produce mutants, which uses a commonly accepted standard set of mutation operators.

As in any empirical analysis, there is the threat to *external validity* regarding the generalization to other types of software. To avoid a bias in the chosen evaluation subjects, we chose five different libraries of very diverse application areas.

7. CONCLUSIONS

Today's techniques for "test" generation actually do not produce tests—they produce sequences of method calls. While they are good at covering code, their effectiveness relies on good run-time checks in the code or the run-time system. To turn these sequences into proper test cases, they need *oracles* that check the correctness of the observed behavior. These oracles need to be effective at detecting faults while being robust against code changes—and most importantly, they need to be confirmed by the developer, which means the entire test case needs to be understandable.

Our approach presented in this paper addresses all of these problems: By systematically exploring and reducing the explicit precondition of a concrete method sequence, we iteratively narrow down the set of relevant preconditions and postconditions, leaving the developer with a crisp representation of the original test. Our initial experiments show that the approach successfully generalizes a concrete method sequence to a parameterized unit test—a test that is more general and more expressive, needs fewer computation steps, and achieves a higher coverage than the original concrete test.

Besides these successes, there are ample opportunities for future work. Generally spoken, the interplay of test generation and specification mining offers plenty of chances for improving both. The combination with specification mining tools such as Daikon will allow us to infer more general properties as pre- and postconditions. Efficient parameterized unit testing tools such as PEX can be used to identify and eliminate false positives and negatives. Our search-based solution in general could be augmented with the use of constraint-based techniques, which might be able to cope with some of the search targets better, and might be able to cut down the overall time to generalize a single test case. Whenever we find an input that does not match the observed behavior, we currently discard it and only retain the information that the currently considered precondition is relevant. This precondition, however, could serve as a splitting point in the specification, leading to an additional PUT which in turn could be optimized. Finally, while we currently focus on unit tests, the is no reason this approach would not scale up to mining specifications for any code.

To learn more about our work in test case generation, visit

`http://www.st.cs.uni-saarland.de/`

Acknowledgments. This project has been funded by Deutsche Forschungsgemeinschaft (DFG), grant Ze509/5-1, and by a Google Focused Research Award on "Test Amplification". Gordon Fraser is funded by the Cluster of Excellence on Multimodal Computing and Interaction at Saarland University, Germany. We thank Eva May, Jeremias Rößler, and the anonymous reviewers for their constructive comments on an earlier version of this paper.

8. REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

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

[6] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, December 2007.

[7] 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.

[8] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *ISSTA'10: Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 147–158. ACM, 2010.

[9] 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.

[10] 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.

[11] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *TAIC-PART '09: Proceedings of Testing: Academic & Industrial Conference - Practice And Research Techniques*, pages 95–104, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[12] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 267–276, Washington, DC, USA, 2005. IEEE Computer Society.

[13] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 417–420, New York, NY, USA, 2007. ACM.

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

[15] 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.

[16] 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.

[17] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 114–123, New York, NY, USA, 2005. ACM.

[18] 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.

[19] 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.

[20] 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.

[21] S. Thummalapenta, M. Marri, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2011)*, 2011.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *the 11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 05)*, pages 365–381, Edinburgh, UK, April 2005.