Generating Tests by Example

Hila Peleg, Dan Rasin, Eran Yahav

Technion {hilap,danrasin,yahave}@cs.technion.ac.il

Abstract. Property-based testing is a technique combining parametric tests with value generators, to create an efficient and maintainable way to test general specifications. To test the program, property-based testing randomly generates a large number of inputs defined by the generator to check whether the test-assertions hold.

We present a novel framework that synthesizes property-based tests from existing unit tests. Projects often have a suite of unit tests that have been collected over time, some of them checking specific and subtle cases. Our approach leverages existing unit tests to learn property-based tests that can be used to increase value coverage by orders of magnitude. Further, we show that our approach: (i) preserves the subtleties of the original test suite; and (ii) produces properties that cover a greater range of inputs than those in the example set.

The main idea is to use abstractions to over-approximate the concrete values of tests with similar structure. These abstractions are then used to produce appropriate value generators that can drive the synthesized property-based test.

We present JARVIS, a tool that synthesizes property-based tests from unit tests, while preserving the subtleties of the original unit tests. We evaluate JARVIS on tests from Apache projects, and show that it preserves these interesting tests while increasing value coverage by orders of magnitude.

1 Introduction

Parametric unit-tests [52, 51, 45, 53, 55] are a well-known approach for increasing coverage and thus increasing confidence in the correctness of a test artifact. Parametric unit tests (PUTs) are also a common target of automatic test generation [22, 7] and unit test generalization [23, 50, 10]. A parametric unit test consists of a *test body* defining the parametric code to execute, and a set of *assumptions* that define the requirements from input values.

Parametric unit tests can either be symbolically executed or *instantiated*, which is the process of turning them back into unit tests [52, 51]. One way to instantiate PUTs is to provide them with concrete values based on whitebox knowledge of the program under test [53, 58]. Another way is to provide a value generator for the parameters, usually hand-crafted by an expert, which generates appropriate values on demand. This type of test is called a *property-based test* (PBT) [19, 11, 16, 30].

I. Dillig and J. Palsberg (Eds.): VMCAI 2018, LNCS 10747, pp. 406–429, 2018.
https://doi.org/10.1007/978-3-319-73721-8_19

The paradigm of property-based testing [13, 4, 1, 43] defines the desired behavior of a program using assertions on large classes of inputs ("property"). To test the program, property-based testing generates inputs satisfying the precondition to check whether the assertion holds. Property-based testing is known to be very effective in checking the general behavior of the code under test, rather than just on a few inputs describing the behavior. It does this by describing the behavior as assertions over classes of input, generating random inputs from that class to check the assertion against. This has the advantages of increasing both instruction coverage and value coverage, and exposing bugs which may be hidden behind the selection of specific representative test cases.

In this paper we present a technique for automatic synthesis of PBTs—parametric unit tests, together with an appropriate value generator—from repetitive unit tests.

The value generators synthesized by our approach follow relationships captured by an abstract representation to explore values within the test's input assumptions. In contrast to the assumptions of parametric unit tests which require a separate enumeration technique (e.g., based on whitebox guidance), abstraction-based generators contain nothing but the definition of the desired input space, and so can be sampled directly and repeatedly to provide a large number of additional values that satisfy the required assumptions.

Our approach generalizes existing unit tests by finding tests with a similar structure such that their concrete values can be over-approximated using an abstract domain. This allows us to use the executed code from the original test, as well as the oracle (assertion) of the test, and execute them with new concrete values. Our approach learns from both positive and negative test-cases (i.e., tests expected to succeed and fail, resp.), enabling a more precise generalization of tests. Specifically, it finds an over-approximation for the positive examples, while excluding any negative examples, and vice versa. In addition, our generalized tests preserve constraints inherent in concrete unit tests, such as types and equalities, which allow us to address the subtle nuances tested by them.

Challenges To achieve our goal, we have to address the following challenges:

- Identify which tests, along with their oracles, should be generalized together to obtain parametric tests.
- Generalize matching tests to find an over-approximation that represents all positive examples but none of the negative ones. This will allow us to synthesize value generators that match the generalized tests.

Existing Techniques Thummalapenta et al. [50] conducted an empirical study analyzing the cost and benefits of manually generalizing unit tests, and have shown that the human effort pays off in increased coverage and newly detected bugs. Shamshiri et al. [46] conducted a test of state-of-the-art test-generators and concluded that they do not create tests as meaningful as human-written tests, leading to the conclusion that basing generalization on existing tests will lead to better results. Fraser and Zeller [23] create tests with pre- and postconditions on parameters, but do so by assuming a baseline version of the program and, in

practice, incorporate its bugs into the tests. Francisco et al. [21] created PBTs for web services, but did so from a semantic description that had to be manually written for each web service. Loscher and Sagonas [34] improve upon PBTs with guided value generation, rather than simple random sampling.

Our Approach The main idea is to leverage the repetitive nature of existing unit tests to automatically synthesize parametric tests and generators. Technically, we define a partial order on the set of tests, that captures the generality of the test data. This order allows our technique to use the same unit test as an example for several different PBTs, capturing different subtleties, and at the same time staves off over-unification of example sets that would yield meaningful results individually, but a non-informative generalization together. We use safe generalization [42] to separate positive and negative examples.

Dividing Tests Provided with individual example unit-tests to be used as a training set, we aim to divide them into sets to be abstracted, in order to create the smallest number of abstractions that are still meaningful, and can still be sampled. We then aim to determine how many value generators are to be created for each such abstracted region of the parameter space. The goal of the division is to create a set of value generators for the property-based tests that will be generated such that each abstracted region can over-approximate the maximal number of examples, and different value generators are created over the same region preserve the testing nuances seen in the original tests. The motivation is that a generator for a PBT must contain the constraints of the subtle cases that were selected by the programmer, to guarantee that these cases are covered in a non-negligible probability when the PBT is executed.

To support this goal, we define a partial order of generality between PBTs. This allows us to create a value generator for each testing nuance, and do so on the maximal number of examples that are compatible with this subtlety.

Safe Generalization of Tests Given a set of compatible positive tests (expected to succeed) and negative tests (expected to fail), we wish to generalize them into a region that a PBT's value generator can sample. To that end, we use an abstraction method for separating positive and negative examples, called *Safe Generalization* [42].

Implementation We present JARVIS (Junit Abstracted for Robust Validation In Scalacheck), a tool that extracts repetitive tests from unit test suites, determines their place in the partial order, and synthesizes from them PBTs that generate inputs based on preserved properties. We test JARVIS on unit tests from Apache projects. We also show that sampling the abstracted over-approximations increases value coverage[8, 29] of the exercised code while not losing instruction coverage. In addition, we demonstrate JARVIS's ability to discover historical bugs when run on test suites in the previous versions.

Main Contributions The contributions of this paper are:

– An inclusion relation between parameterized tests that allows the sharing of examples between different abstracted generators without hindering the ability to abstract.

- A technique that generalizes test values from individual unit tests into value generators for a PBT using safe generalization (separating positive and negative examples).
- A tool, JARVIS, that automatically synthesizes parametric tests, oracles and abstraction-based generators from unit tests, while preserving the subtle cases that are captured in these tests.

2 Overview

Unit tests are an integral part of the software development process. They are used to test small components of large software systems independently. Such components can typically receive many possible inputs, and in order to cover their different behaviors, a component is often run using the same test code with several different input values. In practice this leads to repetitive test code to exercise the same unit under test again and again. An initial study of the repetitiveness in the test suites of five large Apache projects (Commons-Math, Commons-Codec, Collections, Sling and Spark core) showed that of 13,359 total tests, 40% are not unique test scenarios, and 17% are repetitive by being written as an assertion called inside a loop. In some test files, *all* test code is non-unique either by virtue of repetition or loops. This means that repetition of individual tests is not only present but frequent.

However, these tests still use the same values every time the test suite is run. Running identical code with other possible values may reveal a bug, and new bugs may be introduced that will not be tested because of the test values are constant. In fact, tracing through the history of the testing code shows us many such cases: identical tests with a small change of constant values that were later added to represent a bug that has been discovered, and often has been in the code for a full version or more.

We set out to take repetitive test suites and synthesize from them testing properties for property-based testing. Once we have in our possession a parameterized test with an assertion to test its postcondition, as well as a set of values for the parameters labeled for expected success or failure of the test, we can use previous work [47, 49, 33, 20, 17] to learn a precondition on the data and convert it to a data generator for a PBT.

However, dividing test traces into compatible sets is not trivial. Tests may seem to be representing the same case but in their over-unification harm the abstraction. In addition there may be sets of tests that represent an interesting test case, such as equal parameters or a subtype being used, which should be preserved when sampling.

This paper addresses the following problems:

1. Finding individual tests that can be generalized together ("compatible");
2. Generalizing the tests into a property-based test that would cover a superset of the original tests; and
3. Creating abstraction-based value generators that will sample the abstraction while preserving testing nuances.

```
1  Assert.assertTrue(Precision.equals(153.0000, 153.0000, .0625));
2  Assert.assertTrue(Precision.equals(153.0000, 153.0625, .0625));
3  Assert.assertTrue(Precision.equals(152.9375, 153.0000, .0625));
4  Assert.assertFalse(Precision.equals(153.0000, 153.0625, .0624));
5  Assert.assertFalse(Precision.equals(152.9374, 153.0000, .0625));
```

Fig. 1. Several unit tests from the test suite of the Apache commons-math project, using the JUnit testing framework.

To solve 1 we define the notion of tests that are compatible—that test the same thing, and so have the same notion of correctness behind the examples. To solve 2, we use the notion of Safe Generalization in order to find an abstraction that will separate completely the example test cases that are expected to succeed from those that are expected to fail. Finally, to solve 3, we sample these abstractions in a constrained manner dictated by the original tests.

We demonstrate these steps on a real-world example taken from the Apache Commons-Math test suite.

The code segment in Fig. 1 depicts duplicate tests with different constant values in the class `PrecisionTest` in the Commons-Math project. We notice that the seemingly straightforward duplication is not exact duplication. For instance, the test in line 1 uses the same value twice, creating an equality constraint. In fact, in the larger file, there are *several* such tests, using different constants but repeating the value between the first and second parameter.

This means that there is an explicit intention to test the case where the two parameters are equal. Leaving this to chance while drawing reals would make getting two equal values highly unlikely, and the synthesized property would be skipping an intentional special test case if this is not performed. We therefore wish to generate as our output not one but two tests: one for the general case and one for the test with the equality constraint.

Parameterized tests Each test trace is turned into a *parameterized test*. In a parameterized test, constants are extracted and replaced by parameters of the same type. Parameter extraction takes into account constraints that exist in the concrete test, which means that if the same value appears more than once it will be extracted as the same parameter every time. For instance, `assertTrue(Precision.equals(153.0000, 153.0000, .0625));` (line 1) will be parameterized to $pt_1 =$ `assert(Precision.equals(x, x, y));` with types $type(x) = type(y) = double$, and the parameter mapping of $\{x \mapsto 153.0, y \mapsto .0625, res \mapsto +\}$ is preserved, where res signifies the expected result of the assert. Similarly, lines $2-5$ will all be parameterized into $pt_2 =$ `assert(Precision.equals(x, y, z));` with $type(x) = type(y) = type(z) = double$, with four matching parameter mappings.

Grouping parameterized tests into scenarios Parameterized tests that test the same sequence of statements but for the different parameters are grouped together into *scenarios*. All parameterized tests in such a scenario would yield a property-based test that runs the same code, only with differently drawn values

for the parameters. In Fig. 1, both the parameterized tests pt_1 and pt_2 are testing `assert(Precision.equals(?,?,?))` and will be grouped into the same scenario.

All parameterized tests in the same scenario execute the same trace, or in other words test the same thing. A naive solution could use the parameterized data from all test traces belonging to a scenario, and simply perform the abstraction on them, generating a single property-based test for the entire scenario. However, because of the transition from constant values to randomly generated ones, information about the intent of the test is lost. E.g., if the parameterized test sends an integer to a double argument of a method, there is an intent for a number with no fractional part. If the parameterized test repeats a value throughout the test (e.g. between method arguments) there may be an intent for equality. In both cases, the chance of obtaining a value that fits the intention when drawing random values—e.g. from $\mathbb{R}^3$ in the case of Fig. 1—is slim at best.

A simple solution for this could be to keep the tests separated by the parameterized tests that contain them. This means all examples from tests that match `assert(Precision.equals(x,x,y))` with $type(x) = type(y) = double$ will be joined, separate from those that match `assert(Precision.equals(x,y,z))` with $type(x) = type(y) = type(z) = double$. This would generate an additional test forcing the equality of arguments, but would withhold from the unconstrained case with three parameters the additional data points that were separated out. Since both these parameterized tests call the same method, these data points contribute to the understanding of the method's general behavior, and this would cause the generalization of the second test to learn from fewer samples.

A hierarchy of tests A more realistic solution is to abstract as many examples as can be safely unified together, and sample each abstracted region separately later. To do this, we create a hierarchy of parameterized tests based on their parameters. For each parameterized test, we may also consider the data from all the tests below it in the hierarchy. When creating abstractions for the scenario, we consider the maxima of the hierarchy, along with all additional tests that have propagated up to them. This shares as many examples as possible, while preventing over-unification.

To do this, we define an inclusion relation between parameterized tests belonging to the same scenario, based on the sequence of all parameter uses in the test trace. In our example, pt_1 has the parameter sequence $x \cdot x \cdot y$ whereas pt_2 has the parameter sequence $x \cdot y \cdot z$.

We will say pt_1 is a *subtest* of pt_2 because (i) every parameter in place i in the sequence for pt_1 has an implicit conversion to the parameter type of the parameter in place i in the sequence of pt_2, and (ii) any equality constraint in the usage sequence of pt_2 (i.e. the parameter is repeated between places i and j) is also present in the sequence of pt_1. In this case, (i) holds trivially as the types are the same, and (ii) holds because the constraints in pt_1 are relaxed to no constraints in pt_2.

Section 4.2 details the $\sqsubseteq$ relation between two parameterized tests. Section 7 presents experimental data on the importance of using the hierarchical approach.

Abstracting the test data Now that the parameterized tests have been ordered and their concrete samples shared, we can abstract the values of the maxima of the $\sqsubseteq$ relation to a more general behavior. Earlier we parameterized the expected result of the trace with the assignment for res, indicating whether the concrete test should succeed when tested with the constants in the current parameter assignment. This can be used as a label for the parameter assignments as positive or negative examples of the more general property, which we wish to abstract. The examples comprising pt_2 yield the following two sets:

$$\begin{aligned} Positive = \{&(153.0000, 153.0000, .0625),\\ &(153.0000, 153.0625, .0625),\\ &(152.9375, 153.0000, .0625)\}\\ Negative = \{&(153.0000, 153.0625, .0624),\\ &(152.9374, 153.0000, .0625)\} \end{aligned}$$

We are interested in finding an abstraction for the *Positive* and *Negative* sets which explains the partition above, and enables us to generate many more positive and negative examples. It is vital that the abstraction will create a clear-cut separation between the positive and negative examples, in order to ensure that the values drawn will be a superset of the existing examples. This is also a reason that having a large example set is important: having more examples helps grow the abstraction, and having more counterexamples will limit the positive abstraction from covering portions of the input space that should be negative.

To do this, we use the notion of Safe Generalization, and abstract both the positive and negative samples simultaneously, checking that the abstraction of positive examples has not grown to cover negative examples and vice versa.

If there are several maxima in the relation that are being abstracted separately, we notice that the *Negative* set for each of them contains negative examples for the scenario behavior. This means each *Positive* set should be separated from *all Negative* sets, and vice versa. These additional points to be used as counterexamples will improve the separation.

In our case, the abstraction describing the positive examples is $|x - y| \leq z$, and its negation for the negative examples. Section 5 formally defines Safe Generalization and details the use of JARVIS's template library.

In cases where the abstraction is performed on very few samples, there is a lot of room for error for any abstraction. In other programming by example tools such as [28, 31], the solution is to allow the user to mark the solution as incorrect and provide more examples. Section 5.1 discusses the reasons the abstraction may not be ideal and possible solutions.

Sampling the abstraction Once an abstraction is obtained for some set $PT = \{pt_1, pt_2, \ldots, pt_n\}$ of parameterized tests, we turn our attention to sampling the abstracted region, and to the preservation of testing nuances. Because we consider test cases written by the user a weighted sampling of the abstract behavior, we want to make sure we model the sampling of our PBTs in the same fashion.

```
1  val gen_double_1_pos = for(
2    y <- Arbitrary.arbitrary[Double].map(Math.abs);
3    x <- Arbitrary.arbitrary[Double];
4    z <- Gen.choose[Double](x - y, x + y)
5  ) yield (x,y,z)
6  forAll (gen_double_1_pos) {_ match {
7    case (d1: Double,d3: Double,d2: Double) =>
8      Precision.equals(d1, d2, d3)
9  }}
10 val gen_double_1_neg = for(
11   y <- Arbitrary.arbitrary[Double].map(Math.abs);
12   x <- Arbitrary.arbitrary[Double];
13   z <- Gen.oneOf(
14     Gen.choose[Double](Double.MinValue,x - y).suchThat(_ < x - y),
15     Gen.choose[Double](x + y,Double.MaxValue).suchThat(_ > x + y))
16 ) yield (x,y,z)
17 forAll (gen_double_1_neg) {_ match {
18   case (d1: Double,d3: Double,d2: Double) =>
19     !(Precision.equals(d1, d2, d3))
20
21 }}
```

Fig. 2. The ScalaCheck properties synthesized from the test traces shown in Fig. 1.

To do this, we generate an abstraction-based value generator for each $pt_i \in PT$, which will practice *constrained sampling*, i.e., draw values from the abstract region under the parameter constraints of the parameterized test. Section 6 details the way value generators are created over the abstract region.

Finally, we synthesize a PBT to includes each value generator. Fig. 2 shows the resulting properties both the positive and negative data abstractions applied to the concrete samples in $PT = \{pt_1, pt_2\}$, sampled according to pt_2.

Running this property will test the parameterized test on hundreds of values each time. This means that values matching the expected behavior but not covered by the concrete tests will now be tested. This can find bugs that are simply not tested for, and if the test property is added to the test suite, can help stave off bugs that will be added in future changes to the code. Section 8 shows a case study of a historical bug in Apache Commons-Math that was found by using JARVIS on the library's test suite in the version before the bug was corrected.

3 Preliminaries

In this section we introduce concepts used in this paper, including property-based testing and value-based coverage metrics.

Unit test A *unit test* consists of stand-alone code executed against a Unit Under Test (UUT), the result of which is tested against an oracle (an assertion) for correctness. In practice, the code exercising the UUT often targets a small unit, and the oracle is implemented by a set of assertions testing the state and output of the unit test code. Unit testing tools such as JUnit [2] and NUnit [3] provide an environment that can execute an entire *test suite* of unit tests.

Property-based test PBTs consist of test code and an oracle that are defined over parameterized classes of values. For that class of values, the PBT is phrased as a "forall" statement or axiom on the behavior of a component. This means PBTs mirror not a specific code path, but the specifications of the UUT. For example, a simple property on strings would specify that $\forall s_1, s_2, len(s_1 \cdot s_2) = len(s_1) + len(s_2)$.

A property-based test is comprised of two parts: the test body and oracle, which are the code operating on the UUT and the boolean statement which must hold, in this example concatenating and testing the length of strings; and the *generator*, which defines the class of inputs on which the PBT is defined, in this example any two non-null strings.

This is similar to the way *parameterized unit tests* [52] are defined. However, PUTs define the input class by assumptions on the parameters. This means that in order to run as tests in the test suite, PUTs need to be run through a solver or a symbolic execution of the UUT in order to be instantiated with values for the parameters, methods which are usually whitebox. The instantiated parameters are added to the test, which is then transformed into a conventional unit test. Barring a re-run of the solver, the values on which the resulting tests are run are constant.

In contrast, PBTs are intended for execution of the test body on a random sample of values that are drawn from the generator. The generator, rather than describing the input class as a boolean formula (i.e., the conjunction of all assumptions) that filters inputs, defines concretely a portion of the input space from which values can be drawn.

A test using the generator can be added as-is to a test suite using PBT frameworks such as QuickCheck [30, 13], PropEr [41], JSVerify [1] and ScalaCheck [4] that include an initial implementation for the building blocks of generators, such as ScalaCheck's `Gen.choose` used in Fig. 2.

It has been shown [50] that test parametrization is worthwhile in terms of the human effort it requires and the bugs that are detected. It can be extrapolated that PBTs, for which it is easier to draw a large set of test values, would be a worthwhile substitute.

4 Compatible Tests

4.1 From Test Trace to Parameterized Test

In this section, we formally describe how different unit test within a single test suite can be viewed as a repetition of the same test with different parameters. We then continue and formalize what we consider as subtle cases, or testing nuances, appearing is such a group of repetitive tests, and explain how our technique still preserves them.

The first step of our technique is to identify test traces in the original test suite. A *test trace* is a sequence of (not necessarily adjacent) statements ending with a single tested assertion, that can be executed sequentially.

For example, lines 3−4 of Fig. 3 form the test trace `Interval interval = new Interval(2.3,5.7); assertEquals(3.4,interval.getSize());`. Each line in Fig. 1 forms its own test trace, e.g. `assertTrue(Precision.equals(153.0,153.0,.0625));` is formed by line 1.

To handle the many test traces in a library's test suite, we must group them into sets of tests that are compatible for a common abstraction. To this end, we first normalize them and create tests that do not use any specific constant values. This normal form is called a *parameterized test*. Technically, a parameterized test obtained from a test trace contains the same statements as in the test trace, where constant values are replaced by an uninterpreted parameter of the same type as the constant. Moreover, if the same constant appears multiple times in the test trace (at different locations), all occurrences are replaced by the same parameter. Finally, specific assertions such as `assertTrue` or `assertFalse` are replaced with a general `assert` command.

As seen in Section 2, the test trace in line 1 of Fig. 1 is parameterized as $pt_1 =$ `assert(Precision.equals(x, x, y));` with types $type(x) = type(y) = double$. Similarly, the test trace in lines 3−4 of Fig. 3 is parameterized as `Interval interval = new Interval(x,y); assert(z==interval.getSize());` with $type(x) = type(y) = type(z) = double$.

Note that while a concrete test trace holds correctness information (i.e., the desired result of the assertion on a concrete execution of the trace), a parameterized test no longer encodes any such information. The expected result of the assertion is stripped along with the constant values, as it depends on them: the exact same parameterized test might be a positive test on one set of values and a negative test on another.

The relation between a parameterized test and a test trace from which it was originated, relies on the following definition:

Definition 1 (Parameter mapping). *A parameter mapping for a parameterized test is a function f that maps every parameter x to a constant $c = f(x)$ s.t. $type(x) = type(c)$. Additionally, f maps a new variable res to $\{+, -\}$.*

Essentially, a parameter mapping is a function that reproduces the original test trace from a parameterized test. The role of res in the definition above is to represent the type of the assertion (positive or negative). We can think of a test suite as a set of parameterized tests, where each such parameterized test is equipped with a set of parameter mappings $F = \{f_1, \ldots, f_n\}$. Applying each f_i to pt will yield a concrete test trace t_i.

4.2 Separation

Section 5 will explore the abstraction mechanism, but it is easy to see that an abstract representation could be more accurate when working on as large a number of examples as possible. An abstraction that only takes into account the values obtained by the parameter mappings attached to a certain parameterized test may result with a small number of concrete samples. This may yield an abstract

representation which is too coarse. Even worse, the abstract representation may provide no generalization.

To address this, we introduce another definition relying only on the statements in the test trace:

Definition 2 (Scenario). *A* scenario S *is a set of parameterized tests which execute the same sequence of statements, differing only by their parameters.* The code of a scenario S*, is the sequence of statements mutual to all parameterized tests in* S*, after discarding parameter information. We say that a parameterized test pt* belongs *to a scenario* S *if the code of* S *is obtained by discarding pt's parameter information.*

Continuing our example with pt_1 = `assert(Precision.equals(x, x, y));`, if S is the scenario to which pt_1 belongs, then the code of S is the statement `assert(Precision.equals(?, ?, ?));` (without parameter information).

The unification of parameterized tests into scenarios is driven by the fact that despite the different parameter mappings, they are all running the same code (It is important to note that method overloading information is not discarded.)

Next, we formalize subsumption between parameterized tests of the same scenario. These definitions will allow us to increase the number of parameter mappings that can be attached to a single parameterized test.

To define subsumption, we wish to compare two parameterized tests from the same scenario and assess their generality. To do that, we need to compare the parameter uses in the parameterized test in sequence. We therefore rely on the following definition:

Definition 3 (Sequence of parameters). *Given a parameterized test pt, let* $params(pt)$ *be the sequence of parameters across all statements in the parameterized test pt (with repetitions).*

This notion is needed so that we may compare two parameterized tests in the same scenario with a different number of parameters or with equality constraints in different places in the test trace. E.g., for pt =`foo(x,y);assert(bar(x,z));` we have $params(pt) = x \cdot y \cdot x \cdot z$.

Definition 4 (generality of parameterized tests, $\sqsubseteq$). *For two parameterized tests* pt_1, pt_2 *with* $params(pt_k) = x_1^k \cdots x_n^k$ *for* $k \in \{1, 2\}$*, both belonging to the same scenario* S*, we say that* $pt_1 \sqsubseteq pt_2$ *if* $\forall i, j \in \{1 \dots n\}$*:*

1. $type(x_i^1) \sqsubseteq type(x_i^2)$ *(we use the standard notion of this relation, e.g.* $int \sqsubseteq double$*,* $String \sqsubseteq Object$*.)*
2. $name(x_i^2) = name(x_j^2) \Rightarrow name(x_i^1) = name(x_j^1)$

The definition above allows us to create a parameter mapping f_2 for a parameterized test pt_2 from a parameter mapping f_1 for parameterized test pt_1, such that $pt_1 \sqsubseteq pt_2$. We do this by defining the result of f_2 for every $x_i^2 \in params(pt_2)$ by $f_2(x_i^2) = f_1(x_i^1)$.

The implication of the correctness of behavior described by all parameterized tests in a scenario is that all parameter mappings in a scenario can and should

be abstracted together. However, creating a single abstraction for the entire scenario will create a unification problem.

Example 1. Let us consider three parameterized tests that have several parameter mappings each: $pt_1 =$ `int prev = x.size(); x.add(y); assert(x.size() == prev + 1);` with $type(x) = List\langle String\rangle$ and $type(y) = String$, pt_2 is identical to pt_1 except for having $type(x) = ArrayList\langle String\rangle$, and pt_3 is identical to pt_1 except for having $type(x) = Set\langle String\rangle$. Since `add` and `size` are methods on `List` and `Set`'s shared parent interface `Collection`, pt_1, pt_2, and pt_3 all belong to the same scenario.

Since $params(pt_1) = params(pt_2) = params(pt_3) = x \cdot x \cdot y \cdot x$, and since `ArrayList` is a subtype of `List`, but `Set` and `List` only share a common ancestor, we see that $pt_2 \sqsubseteq pt_1$, and pt_3 is incomparable with both.

We notice that even though pt_1 and pt_3 are incomparable, there exists a parameterize test pt_4, with the same test code and $type(x) = Collection\langle String\rangle$ and $type(y) = String$ for which $pt_1 \sqsubseteq pt_4$ and $pt_3 \sqsubseteq pt_4$.

If we aim to abstract pt_4, we can see that our unification problem is twofold. First, we now need to abstract (and later generate values for) collections in general, not just lists and sets, from concrete data that only includes lists and sets. We also see that there is a difference in behavior between sets and lists in this test code which needs to be captured by the abstraction: for a set, $res \mapsto true$ only if y is not already a member of the set, whereas for a list (ArrayList or otherwise), $res \mapsto true$ always. This problem is made even worse in cases where the shared ancestor is $Object$.

In order to avoid these problems we set a unification rule as follows:

Definition 5 (Abstraction candidates). *Let $T \subseteq S$ be the set of parameterized tests in a scenario S such that for every $pt \in T$, $\neg\exists pt' \in S.pt \sqsubseteq pt' \wedge pt \neq pt'$. We define the abstraction candidates for S to be the sets of parameter mappings $AC_S = \{\{f \in pt' \mid pt' \sqsubseteq pt\} \mid pt \in T\}$. When performing abstraction, each $s \in AC_S$ will be abstracted on its own.*

In other words, given the DAG defined by the $\sqsubseteq$ relation, we create an abstraction for every root pt, including with it the parameter mappings of every parameterized test reachable from pt. This means we only create abstractions for parameterized tests that exist "in the wild", whilst reusing as many test traces as possible in order to abstract them.

5 Abstracting the test data

In the following section, we abstract each of the sets of examples in AC_S.

Once a parameterized test has its final set of concrete test traces, the res parameter can be used to divide them into positive and negative samples. For instance, the parameterized test for `Precision.equals(x,y,z)` with $type(x) =$

$type(y) = type(z) = double$ from Fig. 1 has the following data:

$$C^+ = \{(153.0, 153.0, .0625), (153.0, 153.0625, .0625), (152.9375, 153.0, .0625)\},$$
$$C^- = \{(153.0, 153.0625, .0624), (152.9374, 153.0, .0625)\}.$$

Safe Generalization We are interested in an abstraction in *some* language that would be a Safe Generalization [42], or an abstraction that provides separation from a set of counterexamples. Safe Generalization is defined as an operation that further generalizes a set of abstract elements A from an abstract domain [14] into another set of abstract elements, A', while avoiding a set of concrete counterexamples C_{cex}, and provides the following properties:

1. **Abstraction**: A' contains every concrete element that is abstracted by A (even though $A \sqsubseteq A'$ may not hold)
2. **Separation**: No $c \in C_{cex}$ is abstracted by A'
3. **Precision**: Generalization is a direct result of the elements in A.

as well as a strive for **maximality** that is not relevant for this use. We wish to generate two properties for the parameterized test that we are abstracting: one expecting the test to succeed, and one expecting it to fail. The code in these two properties is the same except for a negation of the assertion, but they require different data generators. In order to create these two generators, we need two abstractions, A^+ for the positive examples of the parameterized test, and A^- for the negative.

It is important to notice that, when a scenario has multiple abstraction candidate sets, they are still all representing the same behavior in the code under test, which means they are influenced by the counterexamples in the other sets as well. Specifically, while the positive examples were separated by the unification rule, and should not be abstracted together, they should still be separated from every negative point in the scenario, as they all represent some negative case for the same code. This applies symmetrically to the negative points.

We therefore define for an abstraction candidate $a \in AC_S$ the following example sets:

$$C^+ = \{f \in a | f(res) = +\} \qquad C^- = \{f \in a | f(res) = -\}$$
$$C^+_{cex} = \bigcup_{b \in AC_s} \{f \in b | f(res) = -\} \qquad C^-_{cex} = \bigcup_{b \in AC_s} \{f \in b | f(res) = +\}$$

and attempt to attain the separating abstraction for A^+ from (C^+, C^+_{cex}) and for A^- from (C^-, C^-_{cex}).

It is clear that not every abstraction language will be able to accommodate this requirement. In addition, when there are few samples, many different elements from each language may fit, and we are not necessarily interested in the most precise one, which means we will need to relax the precision requirement of Safe Generalization.

In some domains such as Intervals, we are able to easily compute Safe Generalization using algorithms such as Hydra [36], but we may still wish to perform

a controlled loss of precision on the result. In other domains, computing Safe Generalization will be doubly exponential. Instead, we utilize a Safe Generalization relation, denoted $SG(C, C_{cex})$, which includes the set of abstractions that are safe generalizations for $(\{\beta(c) \mid c \in C\}, C_{cex})$, where β is the abstraction function for a single concrete element. SG relaxes the precision requirement, allowing abstractions to be included in $SG(C, C_{cex})$ even for very small example sets C, C_{cex}.

In theory we would construct SG over every available abstraction. In practice, we use SG to test a set of given abstractions.

Abstraction templates In order to select an abstraction language and an element of that language, JARVIS contains a library of abstraction templates, such as $|x - y| \leq z$, $x \in [a, b]$, etc. As previously shown by the FlashFill project [48], the case of learning from few examples (in FlashFill, often only one example) requires the notion of ranking the possible programs, or in our case, possible abstractions, so that correct programs will be ranked higher than incorrect ones, and likely programs higher than unlikely ones. While in [48] this ranking is learned from examples, in our implementation the templates have a predefined ranking that is applied for all instantiated abstractions that hold for all samples.

Every template t of the template library is instantiated, and in the case of templates such as $x \in [a, b]$ or $|a * x - y| \leq b$, the parameters are selected based on the existing samples. Templates are instantiated in pairs, one as an abstraction for the positive examples and one for the negative. The result is $\mathcal{A} = \{(A^+, A^-) | (A^+, A^-) \in SG(C^+, C^+_{cex}), (A^-, A^+) \in SG(C^-, C^-_{cex})\}$. We then select from $\mathcal{A}$ the highest ranking (A^+, A^-), and create code sampling them as the generators for the properties.

This means the template library can be extended to include more abstractions, and the ranking can be modified to better suit a specific project or domain.

5.1 Handling Impreciseness

The abstractions we use are conservative, and overapproximate the concrete data that they abstract. On the one hand, this guarantees that cases that are present in the original unit test will be included in the generated PBTs. However, in some cases, even the best abstraction available in the template library will be too conservative, and also represent data points that will fail the PBT. This can happen for one of two reasons:

- The abstraction itself is not precise enough (e.g. a single interval, when the data requires a disjunctive abstraction, or a set of intervals).
- The number of examples is too low to precisely generalize from (e.g. generalizing from two examples, there is not enough data to reduce the set of abstraction templates).

Both cases require manual intervention: in the first case, the user can provide a finer abstraction, in the second case she can provide more examples, and in either case she can manually edit the resulting tests.

6 Sampling from the Abstraction

We now wish to sample the abstraction that was created in the previous section. When creating the abstraction-based value generators that will sample the abstraction, we take our cue from the original test traces and their parameterized tests. We consider the original tests written by the programmers to be a weighted sample from the region of the domain that is described by the "true" precondition of the tested behavior. That is, the user has already selected points that they deem important. We therefore wish to preserve them.

We have created an abstraction of each region – an underapproximation of the positive and negative regions for each of the maxima of the $\sqsubseteq$ relation. We now wish to generate property-based tests, or in essence, to generate code that will sample from the concretization of our abstraction. The sampling component of the code in Fig. 2 is shown in lines $1-6$ and $11-17$. It is composed of the representation of the space and types of the variables to be sampled into.

In this section we describe the creation of such sampling for the abstractions we performed.

Sampling based on user-encoded testing nuances We notice that we may wish to sample each abstracted region more than once. Since the constraints of parameterized tests lower in the hierarchy represent constrained values sampled by the user, we wish to cover them in our generated sampling. Let us examine the parameterized test $pt_1 =$ `assert(Precision.equals(x, x, y));` with $type(x) = type(y) = double$ from Section 2. It is sampled out of the region abstracted for the entire scenario S containing pt_1 as well as other tests for `Precision.equals`. Abstracting the topmost parameterized test by the $\sqsubseteq$ relation yields an abstraction in $\mathbb{R}^3$. When sampling $(x, y, z) \in A \subseteq \mathbb{R}^3$ the odds of satisfying the constraint in pt_1, i.e., $x = y$, are infinitesimal. If we wish to preserve the constraint entered by the user, we must sample the special case in which $x = y$ on its own.

Sampling the constraints In order to sample each set of constraints on its own, we create a sampling component as follows: for every $pt \in S$, we create a sampling component over each abstraction for an abstraction candidate s for which pt has contributed its parameter mappings. For each region sampled, the constraints of pt are added to the restrictions on the domain. For example, when sampling the region $|x - y| \leq z$ for pt_1 seen in Section 2, the new sampling constraints are $|x - y| \leq z \wedge x = y$, or $0 \leq z$, sampling (x, z) out of this region s.t. $type(x) = type(z) = double$.

Sampling guarantee Finally, we formulate our guarantee for points that will be sampled:

Claim. Let T be a set of test traces from the same scenario S, $|T| \geq 2$. For each $t \in T$, if $\exists t' \neq t$ s.t. $PT(t) \sqsubseteq PT(t')$, then

1. t will be used in an abstraction, and
2. $PT(t)$ will be used to create an abstraction-based value generator for a PBT.

	Scenarios		Repeating scenarios					Hierarchy			
			no. of traces		no. PTs per scenario		have multiple	height		no. of roots	
Library	avg size	repeating scenarios	avg	max	avg	max	PTs	avg	max	avg	max
Commons-CLI	4.2	38.3%	3.5	14	1.067	2	6.7%	1.033	2	1.50	2
Commons-Codec	2.2	38.2%	3.4	8	1.088	2	8.8%	1.088	2	1.00	1
Commons-Collections	2.1	13.3%	2.8	5	1.133	4	6.7%	1.033	2	2.50	3
Commons-Configuration	3.5	14.6%	3.7	15	1.015	2	1.6%	1.012	2	1.25	2
Commons-CSV	3.2	27.8%	2.0	2	1	1	0.0%	1.000	1	1.00	1
Commons-Email	2.3	60.0%	2.7	5	1.1	2	10.0%	1.100	2	1.00	1
Commons-IO	2.9	23.8%	3.6	14	1.069	4	4.7%	1.042	2	1.11	2
Commons-JEXL	4.6	25.9%	2.4	4	1.037	2	3.7%	1.000	1	2.00	2
Commons-Lang	2.1	36.7%	5.5	37	1.273	5	19.1%	1.212	3	1.04	3
Commons-Math	4.3	19.7%	4.1	45	1.182	9	10.2%	1.075	4	1.79	6
Commons-Pool	3.9	33.3%	2.0	2	1.222	2	22.2%	1.222	2	1.00	1
Commons-Text	2.5	36.7%	6.2	15	1.133	2	13.3%	1.133	2	1.00	1

Table 1. Scenario makeup of the JUnit test suites of Apache-Commons projects. Repeating scenarios are those with the number of concrete test traces greater than 1.

This allows for a maximal reuse of examples for abstraction, and on the other hand, the sampling of all special cases that are abstracted.

7 Experimental evaluation

We implemented JARVIS to operate on JUnit test suites written in Java and to synthesize ScalaCheck PBTs. Scala has a seamless interoperability with Java [38], which means properties for ScalaCheck, which are written in Scala, can mimic completely the functionality of the original test traces. JARVIS uses the Polyglot compiler [37] and the ScalaGen [5] project to translate test traces from Java to Scala. Template instantiation is aided by the Z3 SMT solver [15].

We ran JARVIS on the test suites of several open source libraries. We tested whether the hierarchy and unification rule of abstraction candidates are relevant to real-world test suites.

7.1 Examining Apache test suites

Tab. 1 shows the result of running JARVIS on the test suites of 12 Apache Commons projects. This summary of JARVIS's ability to unify shows us several things in regard to the problems it addresses:

Identifying tests In all projects, the average length of extracted test traces is over two statements. This shows JARVIS identifies and extracts elaborate tests.

Repetition of tests The data shows that there is, in fact, enough repetition of tests to justify test generalization. In the project with the least amount of repetition, Commons-Collections, only 13% of the scenarios contain more than one concrete test trace, but in half the projects this number is over 30%.

Existence of constraints When examining the scenarios that contain more than one test trace, we see that this is, on average, 9% of scenarios, with as many as 9 separate parameterized tests, or sets of constraints, in the same scenario.

These are all user-encoded sampling constraints that would see their probability plummet without a specific sampler generated for them.

Importance of the unification rule In scenarios that have multiple parameterized tests, we see that the number of roots in the hierarchy DAG (i.e., incomparable maxima of the relation) is, on average, 1.35. This means that it is not infrequent to have scenarios where the least upper bound of two or more parameterized tests does not occur in the test traces. Since this is a frequent occurrence in the real world, we deem it frequent enough to address the issues that arise from over-unification within the scenario (as described in Section 4.2).

7.2 Increased coverage

The extended version of our paper contains coverage experiments comparing JARVIS-generated PBTs to the original test suite from which it was generated. Scenarios from Apache Commons-Math and Commons-Lang were included.

Instruction coverage, or the number of lines of code in the library under test that were exercised by the test code, was preserved or marginally improved for all benchmarks. *Value coverage*, however, was sometimes increased by up to two orders of magnitude. This increase is especially important because of the ability to find bugs that are not "boundary values" in terms of the structure of the program, like the one described in Section 8.

Additionally, while running these experiments we managed to identify a bug in the Common-Lang test suite, and our submitted repair[1] was accepted.

8 Discovering Bugs: A Case Study

In this section we review a historical bug in Apache Commons-Math that we discovered by running JARVIS on the unit test suite for the version before the bug was fixed.

The extended version of this paper includes a second bug found by JARVIS: a critical severity bug in the `ContinuedFraction` class, discovered via the PBT generated from the unit tests for the `FDistribution` class[2]. Unlike the full case study presented here, this bug required manual intervention as described in Section 5.1.

8.1 MATH-1256: Interval bounds

In Apache Commons-Math versions prior to 3.6, the test suite for the `Interval` class included the code in Fig. 3. A bug in the interval class which is not tested in these unit tests was opened as "MATH-1256: Interval class upper and lower check"[3]. An Interval object could be created with a lower bound greater than its

[1] http://github.com/apache/commons-lang/pull/230
[2] http://issues.apache.org/jira/browse/MATH-785
[3] http://issues.apache.org/jira/browse/MATH-1256

```
@Test
public void testInterval() {
    Interval interval = new Interval(2.3, 5.7);
    Assert.assertEquals(3.4, interval.getSize(),
        1.0e-10);
    Assert.assertEquals(4.0, interval.getBarycenter(),
        1.0e-10);
    Assert.assertEquals(Region.Location.BOUNDARY,
        interval.checkPoint(2.3, 1.0e-10));
    //Other asserts on properties of interval
}
//...
@Test
public void testSinglePoint() {
    Interval interval = new Interval(1.0, 1.0);
    Assert.assertEquals(0.0, interval.getSize(),
        Precision.SAFE_MIN);
    Assert.assertEquals(1.0, interval.getBarycenter(),
        Precision.EPSILON);
}
```

Fig. 3. Unit test code for the Interval class from the Apache Commons-Math project

```
val gen_double_1_pos  = for(
    x <- Gen.posNum[Double];
    y <- Arbitrary.arbitrary[Double];
    z <- Gen.oneOf(y-x,y+x)
        .suchThat(t => Math.abs(t-y) == x)
    ) yield (x,y,z)

forAll (gen_double_1_pos) {_ match {
case (double_1,double_3,double_2) =>
  val interval = new Interval(double_1, double_2)
  double_3 ~= interval.getSize
}}
```

Fig. 4. The ScalaCheck generator and property generated by JARVIS from the unit tests in Fig. 3.

upper bound, which would result in an invalid interval with a negative size. The bug report shows test code initializing `Interval interval0 = new Interval(0.0, (-1.0));` and showing that it would result in `interval0.getSize()` being -1.0.

This bug hinges on the two parameters accepted by the `Interval` constructor. Since it only requires $y > x$, it exists in nearly 50% of the parameter space. However, the conventional unit tests in `IntervalTest` only cover 2 values.

Running JARVIS on `IntervalTest.java` from release 3.5 yields 9 different scenarios. The scenario testing `getSize` contains two parameterized tests, one for the parameters $double_1$, $double_2$ and $double_3$, from the code in `testInterval` and one for $double_1$ and $double_2$ from the code in `testSinglePoint`. We denote them pt_3 and pt_4, respectively. Since $pt_4 \sqsubseteq pt_3$, the concrete test from `testSinglePoint` is added to the parameterized test for `testInterval`, resulting in $C^+ = \{(2.3, 5.7, 3.4), (1.0, 1.0, 0.0)\}$.

From the abstraction template library for 3D abstractions, the abstraction selected for these points by the criteria outlined in Section 5 is $|y - z| = x$.

JARVIS outputs the code in Fig. 4 to generate values matching the abstraction. Running the test with ScalaCheck fails in cases where the upper bound of the interval is negative while the lower bound is generated as always positive. Since the bug exists in nearly 50% of the space, it occurs almost immediately when running the PBT. These cases expose MATH-1256 without the additional unit tests that were later added after it was reported and fixed.

9 Related Work

Learning from examples Learning from examples or "Programming by Example" is a field of synthesis with many different applications, such as Inductive Programming [18], string processing [28, 27, 48] and data extraction [31]. In particular, the FlashFill and FlashExtract projects [48, 31] present an interactive algorithm for synthesis by examples used to generate code for string manipulation, showing that it is possible to synthesize a program from few examples, despite having several compatible solutions.

Genrating Tests and Oracles An experiment described in [46] reveals that state of the art automatic test generation tools are far from satisfactory. However, an experiment described in [44] shows that manual unit tests, written by developers aided by an automatic test generation tool, create better code coverage. [40] use code instrumentation of the system under test to guide test generation by path discovery. [39] suggest a technique that improves random test generation by avoiding sequences calls after an object has reached an illegal state. [57] extend this technique further to increase coverage and diversity for automatically generated tests. However, their method only generates the tests and they do not suggest how to generate oracles. [22] describe a technique for automatically generating unit tests together with appropriate pre and post-conditions, based on mutations of the tested class and test inputs. However, the basis for the postcondition is the observable state after the test execution, which means that bugs in the program will result in incorrect postconditions. [24] suggests a mutation-based technique to select variables for which an oracle would detect a high number of faults in the program. However, a tester is still required to write the oracles. [58] generate unit tests using symbolic execution and incremental refinement. [54] generate both tests and oracles from use case specifications, using natural language processing techniques.

Parameterized Unit Tests PUTs are defined in [52] and as "theory-based testing" in [45] and developed further in [51, 53, 55]. [50] is an empirical study in unit test parametrization that strongly advocates parameterized unit testing. Generalizing unit tests to parameterized unit tests was shown as useful in detecting new bugs and required feasible human effort, though one that required expertise with additional tools. However, their proposed methodology contains manual steps for parametrization and generalization, and they do not address the problem of extracting and grouping the tests. [23] extends [22] to parameterized unit tests, but exactly as in [22], the postcondition is derived from the

observable state after the test execution. In contrast, JARVIS creates test oracles from the oracles of the original unit tests, and treats their assertions as part of the generalization, making no assumptions based on the execution.

Property Based Testing and Fuzzing [21] creates PBTs for web services, but does so from both a syntactic and manually-written semantic description of the service. Later work [32] is intended to track API changes in web services and update existing PBTs. [25] recognizes the important connection between conventional unit tests and PBTs, and describe a tool that checks whether a given unit test is covered by a given PBT. [56] is a tool for automatic PBT generation, based on feedback directed random test generation [39]. However, similar to previous feedback directed random test generation techniques, the oracles are specified by the developer. Fuzz testing or "fuzzing", another testing technique, is very similar to property-based testing: it draws inputs or enumerates them, but usually does not use oracles, only looks for crashes, and does not test components but rather the whole program. Works such as [9] draw their inputs from grammars of valid or invalid inputs. Others add to this a white-box approach [26, 35], attempting to draw inputs that increase coverage. [12] suggest combining fuzzing with ranking, based on the diversity of the test cases. [6] aim to draw inputs for fuzzing that will direct the fuzzing to a part of the program.

10 Conclusion

We presented JARVIS, a tool to extract repetitive tests from unit test suites and synthesize from them property-based tests. We have shown the foundations for its operation: sorting the existing unit tests into sets of compatible tests; a better abstraction, achieved using a hierarchy of generality between parameterized tests which allows abstractions to generalize more tests; generalizing the examples to a data generator, taking into consideration the positive and negative examples (tests expected to succeed and fail, resp.) using the notion of Safe Generalization; and preserving the subtleties of human-written unit tests by sampling each abstracted region according to constraints found in the test data.

We applied JARVIS to the JUnit test suites of 12 Apache Commons APIs, and have shown there is ample repetition in the data of real-world test suites, which can be used to generate PBTs. We have also shown that the repetition often includes subtleties, in the same testing scenario. Additionally, we have shown that JARVIS-generated PBTs maintain the instruction coverage of the original unit tests, and increase parameter value coverage by as much as two orders of magnitude. PBTs generated by JARVIS have found a known bug in Apache Commons-Math, and with the help of JARVIS we identified a bug in the Commons-Lang test suite.

Acknowledgements

The research leading to these results has received funding from the European Unions Seventh Framework Programme (FP7) under grant agreement no. 615688 - ERC-COG-PRIME.

References

1. Jsverify: Write powerful and concise tests.
2. Junit. http://junit.org/.
3. Nunit. http://www.nunit.org/.
4. Scalacheck: Property-based testing for scala. http://www.scalacheck.org/.
5. Scalagen: Java to scala conversion.
6. M. A. Alipour, A. Groce, R. Gopinath, and A. Christi. Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 70–81. ACM, 2016.
7. D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina. Automatic test generation for coverage analysis using cbmc. In *International Conference on Computer Aided Systems Theory*, pages 287–294. Springer, 2009.
8. T. Ball. A theory of predicate-complete test coverage and generation. In *International Symposium on Formal Methods for Components and Objects*, pages 1–22. Springer, 2004.
9. R. Brummayer and A. Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5. ACM, 2009.
10. J. R. Calamé, N. Ioustinova, and J. van de Pol. Automatic model-based generation of parameterized test cases using data abstraction. *Electronic Notes in Theoretical Computer Science*, 191:25–48, 2007.
11. M. Carlier, C. Dubois, and A. Gotlieb. Constraint reasoning in focaltest. In *ICSOFT*, 2010.
12. Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 197–208, New York, NY, USA, 2013. ACM.
13. K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.
14. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
15. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
16. J. Derrick, N. Walkinshaw, T. Arts, C. B. Earle, F. Cesarini, L.-A. Fredlund, V. Gulias, J. Hughes, and S. Thompson. Property-based testing-the protest project. In *International Symposium on Formal Methods for Components and Objects*, pages 250–271. Springer, 2009.
17. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001.
18. C. Ferri-Ramírez, J. Hernández-Orallo, and M. Ramírez-Quintana. Incremental learning of functional logic programs. In *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS'01)*, volume 2024 of *LNCS*, pages 233–247. Springer-Verlag, 2001.
19. G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.
20. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal*

Methods for Increasing Software Productivity, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.
21. M. A. Francisco, M. López, H. Ferreiro, and L. M. Castro. Turning web services descriptions into quickcheck models for automatic testing. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, pages 79–86. ACM, 2013.
22. G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 147–158, New York, NY, USA, 2010. ACM.
23. G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 364–374. ACM, 2011.
24. G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl. Automated oracle data selection support. *Software Engineering, IEEE Transactions on*, 41(11):1119–1137, 2015.
25. A. Gerdes, J. Hughes, N. Smallbone, and M. Wang. Linking unit tests and properties. In *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang*, pages 19–26. ACM, 2015.
26. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.
27. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.
28. S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, Aug. 2012.
29. D. Hoffman, P. Strooper, and L. White. Boundary values and automated component testing. *Software Testing, Verification and Reliability*, 9(1):3–26, 1999.
30. J. Hughes. Quickcheck testing for fun and profit. In *International Symposium on Practical Aspects of Declarative Languages*, pages 1–32. Springer, 2007.
31. V. Le and S. Gulwani. FlashExtract: a framework for data extraction by examples. In M. F. P. O'Boyle and K. Pingali, editors, *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 55. ACM, 2014.
32. H. Li, S. Thompson, P. Lamela Seijas, and M. A. Francisco. Automating property-based testing of evolving web services. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, pages 169–180. ACM, 2014.
33. N. P. Lopes and J. Monteiro. Weakest precondition synthesis for compiler optimizations. In *Verification, Model Checking, and Abstract Interpretation*, pages 203–221. Springer, 2014.
34. A. Löscher and K. Sagonas. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 46–56. ACM, 2017.
35. R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 134–143. ACM, 2007.
36. K. S. Murray. Multiple convergence: An approach to disjunctive concept acquisition. In *IJCAI*, pages 297–300. Citeseer, 1987.
37. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction*, pages 138–152. Springer, 2003.
38. M. Odersky, L. Spoon, and B. Venners. Scala. *URL: http://blog. typesafe. com/why-scala (last accessed: 2012-08-28)*, 2011.

39. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.
40. R. Pandita, T. Xie, N. Tillmann, and J. De Halleux. Guided test generation for coverage criteria. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.
41. M. Papadakis and K. Sagonas. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pages 39–50. ACM, 2011.
42. H. Peleg, S. Shoham, and E. Yahav. D3: Data-driven disjunctive abstraction. In *Verification, Model Checking, and Abstract Interpretation*, pages 185–205. Springer, 2016.
43. L. Pike. Smartcheck: automatic and efficient counterexample reduction and generalization. In *ACM SIGPLAN Notices*, volume 49, pages 53–64. ACM, 2014.
44. J. M. Rojas, G. Fraser, and A. Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 338–349, New York, NY, USA, 2015. ACM.
45. D. Saff, M. Boshernitsan, and M. D. Ernst. Theories in practice: Easy-to-write specifications that catch bugs. 2008.
46. S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 201–211. IEEE, 2015.
47. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification*, pages 88–105. Springer, 2014.
48. R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *Computer Aided Verification*, pages 398–414. Springer, 2015.
49. S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *ACM Sigplan Notices*, volume 44, pages 223–234. ACM, 2009.
50. S. Thummalapenta, M. R. Marri, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, FASE'11/ETAPS'11, pages 294–309, Berlin, Heidelberg, 2011. Springer-Verlag.
51. N. Tillmann and P. de Halleux. Parameterized unit testing with pex (tutorial). In *Proc. of Tests and Proofs (TAP'08)*, volume 4966, page 171181, Prato, Italy, April 2008. Springer Verlag.
52. N. Tillmann and W. Schulte. Parameterized unit tests. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 253–262. ACM, 2005.
53. N. Tillmann and W. Schulte. Parameterized unit tests with unit meister. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 241–244. ACM, 2005.
54. C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal. Automatic generation of system test cases from use case specifications. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 385–396, New York, NY, USA, 2015. ACM.
55. T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mutation analysis of parameterized unit tests. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 177–181. IEEE, 2009.

56. K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. Arbitcheck: A highly automated property-based testing tool for java. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '14, pages 405–412, Washington, DC, USA, 2014. IEEE Computer Society.
57. K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. Feedback-controlled random test generation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 316–326, 2015.
58. H. Yoshida, S. Tokumoto, M. R. Prasad, I. Ghosh, and T. Uehara. Fsx: Fine-grained incremental unit test generation for c/c++ programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 106–117. ACM, 2016.