Teralizer: Semantics-Based Test Generalization from Conventional Unit Tests to Property-Based Tests

JOHANN GLOCK, University of Klagenfurt, Austria
CLEMENS BAUER, University of Klagenfurt, Austria
MARTIN PINZGER, University of Klagenfurt, Austria

Conventional unit tests validate single input-output pairs, leaving most inputs of an execution path untested. Property-based testing addresses this shortcoming by generating multiple inputs satisfying properties but requires significant manual effort to define properties and their constraints. We propose a semantics-based approach that automatically transforms unit tests into property-based tests by extracting specifications from implementations via single-path symbolic analysis. We demonstrate this approach through Teralizer, a prototype for Java that transforms JUnit tests into property-based jqwik tests. Unlike prior work that generalizes from input-output examples, Teralizer derives specifications from program semantics.

We evaluated Teralizer on three progressively challenging datasets. On EvoSuite-generated tests for EqBench and Apache Commons utilities, Teralizer improved mutation scores by 1–4 percentage points. Generalization of mature developer-written tests from Apache Commons utilities showed only 0.05–0.07 percentage points improvement. Analysis of 632 real-world Java projects from RepoReapers highlights applicability barriers: only 1.7% of projects completed the generalization pipeline, with failures primarily due to type support limitations in symbolic analysis and static analysis limitations in our prototype. Based on the results, we provide a roadmap for future work, identifying research and engineering challenges that need to be tackled to advance the field of test generalization.

Artifacts available at: https://doi.org/10.5281/zenodo.17950380

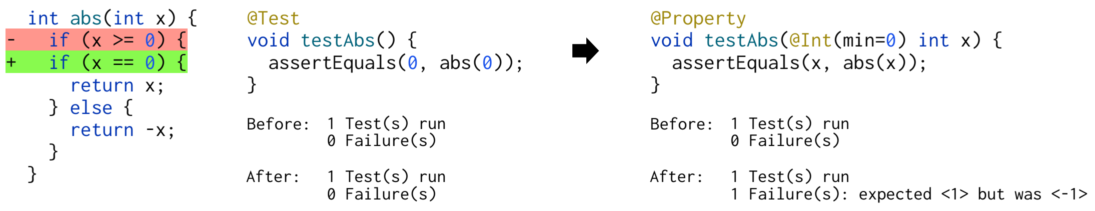
CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; Empirical software validation; • **Theory of computation** → *Program analysis*.

Additional Key Words and Phrases: Test Amplification, Test Generalization, Property-Based Testing, Symbolic Execution

1 Introduction

Conventional unit tests validate software behavior by checking specific input-output pairs [2, 56, 68], but leave most inputs along the same execution path untested. Property-based testing [15, 37] instead generates many inputs and checks whether specified properties hold across executions. For example, given the *abs* method in Figure 1, a unit test which asserts that $abs(0)$ returns 0 would still pass after changing `x >= 0` to `x == 0`, whereas a property-based test which asserts $abs(x) = x$ for $x \geq 0$ would expose this regression. Industrial experience reports suggest that property-based testing often uncovers edge cases and boundary conditions missed by unit tests [33, 38]. Adoption, however, remains limited because writing property-based tests requires manual effort to define both input constraints and suitable properties, a task practitioners find challenging [33]. This motivates research into transformation approaches that automatically generalize existing unit tests by deriving properties from program semantics.

Authors' Contact Information: Johann Glock, johann.glock@aau.at, University of Klagenfurt, Klagenfurt, Austria; Clemens Bauer, clemens.bauer@aau.at, University of Klagenfurt, Klagenfurt, Austria; Martin Pinzger, martin.pinzger@aau.at, University of Klagenfurt, Klagenfurt, Austria.

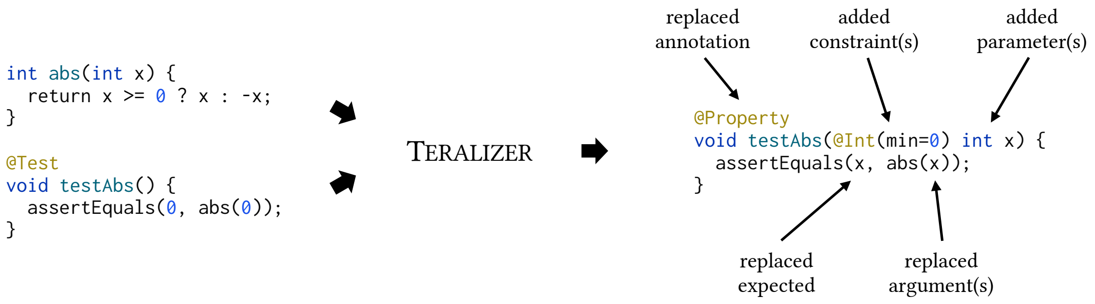


```
  int abs(int x) {
-   if (x >= 0) {
+   if (x == 0) {
      return x;
    } else {
      return -x;
    }
  }
```

```
@Test
void testAbs() {
  assertEquals(0, abs(0));
}

Before:  1 Test(s) run
         0 Failure(s)

After:   1 Test(s) run
         0 Failure(s)
```

```
@Property
void testAbs(@Int(min=0) int x) {
  assertEquals(x, abs(x));
}

Before:  1 Test(s) run
         0 Failure(s)

After:   1 Test(s) run
         1 Failure(s): expected <1> but was <-1>
```

Fig. 1. The conventional unit test misses a regression that the property-based test detects.



```
int abs(int x) {
  return x >= 0 ? x : -x;
}

@Test
void testAbs() {
  assertEquals(0, abs(0));
}
```

```
@Property
void testAbs(@Int(min=0) int x) {
  assertEquals(x, abs(x));
}
```

Fig. 2. TERALIZER takes implementation and test code as input, and produces property-based tests as output.

We propose a semantics-based approach for automated test generalization that analyzes both test and implementation code to derive path-exact specifications through single-path symbolic analysis [64]. Our method determines which inputs follow the same execution path as existing tests and transforms unit tests into property-based tests that validate the same assertions across entire input partitions. Because specifications are extracted directly from program semantics, the resulting properties are exact for each execution path and preserve the developer-provided oracles encoded in assertions. To our knowledge, JARVIS [66] is the only prior work that automatically generalizes unit tests into property-based tests. However, JARVIS infers properties from input-output examples based solely on test code, relying on predefined abstraction templates that yield overapproximations. In contrast, our white-box approach leverages both static and dynamic program analysis to extract exact specifications for the execution paths exercised by the original tests.

We implemented this approach in TERALIZER, a prototype tool for Java that transforms JUnit tests into property-based jqwik [41] tests. TERALIZER employs a five-stage pipeline: (1) analyzing tests and their assertions regarding suitability for generalization, (2) identifying tested methods through data flow analysis, (3) extracting specifications through single-path symbolic analysis [64], (4) creating generalized property-based tests, and (5) filtering generalized tests to retain only those that improve fault detection capability. Figure 2 illustrates the effects of this transformation, showing how a simple equality assertion $abs(0) = 0$ becomes the property $abs(x) = x$, valid for all non-negative values of x.

To evaluate our approach, we applied TERALIZER to three complementary datasets. The EQBENCH benchmark [4] provides controlled settings with numeric-focused programs well-suited for symbolic analysis. Because EQBENCH lacks test suites, we generated tests using EVOSUITE [29]. Utility methods extracted from Apache Commons projects offer a middle ground between controlled and real-world scenarios. Here, we directly compared EVOSUITE-generated and developer-written tests on the same codebase, partially isolating the influence of test architecture on generalization outcomes. Finally, we applied Teralizer to 632 real-world Java projects with developer-written tests from the RepoReapers dataset [55] to expose the full complexity of practical application scenarios. This progression from controlled to real-world conditions highlights both the potential and limitations of semantics-based test generalization.

Our evaluation shows modest yet consistent improvements under controlled conditions. On EvoSuite-generated tests, mutation scores increased by 1–4 percentage points: from 48–52% to 52–55% on EqBench, and from 57–58% to 58–59% on Apache Commons utilities. In contrast, generalization of developer-written tests for Apache Commons utilities showed only 0.05–0.07 percentage points improvement from a baseline of 80.35%. Results from the RepoReapers projects reveal practical applicability barriers: only 1.7% of projects successfully completed the generalization pipeline. Failures primarily occurred due to type support limitations of symbolic analysis as well as static analysis limitations of our prototype. To provide a roadmap for future work, we classified these failures into those that can be resolved through engineering effort and those that represent deeper research challenges in specification extraction and encoding.

This paper makes the following contributions:

(1) A **semantics-based test generalization approach** that extracts specifications via symbolic analysis to transform conventional unit tests into property-based tests.
(2) A comprehensive **empirical evaluation** across three complementary datasets, demonstrating 1–4 percentage point mutation score improvements under controlled conditions.
(3) A systematic **analysis of applicability barriers**, distinguishing addressable engineering limitations from fundamental research challenges in specification extraction and encoding.
(4) An **open implementation and replication package** [32], enabling reproduction and extension of our results.

The paper is organized as follows. Section 2 introduces the technical foundations of test generalization. Section 3 presents Teralizer's five-stage pipeline. Section 4 evaluates our approach through six research questions, covering mutation score improvements, impact on test suite size and execution time, runtime requirements, and causes of unsuccessful generalizations. Section 5 discusses the results, directions for future work, and threats to validity. Section 6 positions our work within the broader testing literature, and Section 7 concludes the paper.

2 Background

This section provides the technical foundations for semantics-based test generalization. Section 2.1 situates our work within the test amplification landscape. Section 2.2 introduces property-based testing as our target representation. Section 2.3 describes single-path symbolic analysis, the technique we use for specifications extraction. Finally, Section 2.4 presents mutation testing as our evaluation methodology for assessing the effectiveness of generalized tests.

2.1 Test Amplification and Generalization

Test amplification uses knowledge embedded in implementations and tests of software projects to automatically enhance the projects' test suites. Danglot et al.'s taxonomy [21] distinguishes four categories of amplification: AMP_{add} creates new tests from existing ones, $\text{AMP}_{\text{change}}$ targets specific program modifications, AMP_{exec} varies execution conditions, and AMP_{mod} modifies test structure or assertions to generalize behavior. Test generalization belongs to the AMP_{mod} category. It transforms tests from validating individual input-output pairs to validating properties across entire input partitions. For example, a test which asserts that $abs(0)$ returns 0 validates abs for only a single input-output pair, missing regressions that preserve the behavior at that point but violate the general property $abs(x) = x$ which should hold when $x \geq 0$ (Figure 1). Since this property is implicitly encoded in the original test, we can automatically transform the test into a corresponding property-based test (Figure 2). A central challenge in test amplification approaches is the oracle problem: determining expected outputs for new test inputs [6]. Existing tests provide validated oracles for their specific execution paths, encoding developer knowledge about expected behavior. Other execution paths lack equally trustworthy oracles, making it difficult to distinguish intentional behavior from incidental state changes or outputs.

2.2 Property-Based Testing as Target Representation

Property-based testing (PBT) — pioneered by QuickCheck for Haskell [15] and now available via, e.g., ScalaCheck for Scala [58], Hypothesis for Python [48], and jqwik for Java [41] — validates specifications over input partitions rather than single inputs. PBT frameworks comprise three key components. *Generators* produce inputs according to specified constraints, such as $x \geq 0$. *Properties* express invariants that must hold for all generated inputs, such as $abs(x) = x$ for non-negative x. *Shrinking* minimizes failing inputs to simplify debugging, such as reducing the input 776,837 to 1. This generative approach distinguishes PBT from parameterized testing, which commonly relies on predefined inputs.

For example, the property-based test in Figure 1 uses `@Property` to indicate property-based testing and `@Int(min=0)` to constrain input generation to non-negative integers. It then validates that `assertEquals(x, abs(x))` holds for all generated values. When this test executes, jqwik generates hundreds of non-negative integers, including edge cases like `0`, `1`, and `Integer.MAX_VALUE`, and checks that the property holds for each one. If a failure occurs, the framework's shrinking algorithm automatically reduces the failing input to its minimal form, simplifying debugging.

The combination of constrained generation and property checking enables thorough exploration of input spaces, revealing edge cases that developers might not explicitly consider [38, 48]. However, adoption of property-based testing remains limited [33]. Creating property-based tests requires identifying appropriate properties, defining input generators with suitable constraints, and translating example-based assertions into general specifications: a conceptual shift that can be difficult for developers [6, 79] even though conventional unit tests already encode behavioral properties implicitly: an assertion $abs(0) = 0$ reflects the property $abs(x) = x$ for $x \geq 0$ but validates it only for a single input.

2.3 Symbolic Analysis for Specification Extraction

Automating the transformation from conventional tests into property-based specifications requires extracting two elements: the *path condition* that characterizes inputs following the same execution path, and the *symbolic output expression* that computes expected results for those inputs. For example, the $abs(0)$ test in Figure 2 yields the path condition $x \geq 0$ and the symbolic output x. These path-exact specifications enable property-based tests that validate behavior across entire input partitions while preserving the original test's semantics.

Single-path symbolic analysis achieves this extraction by following the concrete execution path of an existing test while maintaining symbolic representations of variables [64]. Unlike full symbolic execution, which faces path explosion when exploring all possible paths [5, 12], single-path analysis omits backtracking and constraint solving, recording conditions only along the executed path. This focused approach is well suited to test generalization: existing tests identify the behaviors of interest and provide validated oracles for those behaviors.

The precision of extracted specifications depends strongly on the data types of the involved variables. Linear integer constraints (e.g., $x > 0$, $y \leq 2 \cdot x$) are well supported by symbolic execution tools such as Symbolic PathFinder (SPF) for Java [64] and KLEE for C [11]. Non-linear arithmetic and floating-point operations are more problematic: while tools can still represent them precisely, constraint solving quickly becomes computationally intractable, leading to timeouts [23]. Strings, arrays, and complex objects pose the greatest practical barrier: symbolic representations typically lose precision or become overly abstract, limiting their usefulness for specification extraction [1, 5].

These limitations affect test generalization at two distinct stages. First, imprecise specifications (as with complex types) prevent generalization entirely since we cannot create meaningful property-based tests without accurate models. Second, even with precise specifications (as with non-linear numeric constraints), test generalization succeeds but the resulting tests may fail during execution when PBT frameworks cannot produce inputs satisfying complex constraints. This input generation difficulty represents a fundamental computational challenge that frameworks cannot overcome through filtering or constraint encoding [15, 41]. Thus, while test generalization can theoretically handle any accurately modeled behavior, practical success requires both precise specifications and tractable constraints.

2.4 Mutation Testing for Evaluation

Having established how to extract specifications from existing tests, we require a systematic way to assess whether transforming tests based on these specifications improves fault detection capability. Because original and generalized tests execute the same paths, traditional coverage metrics cannot reveal improvements [39]. Statement, branch, and path coverage [87] remain identical whether a test validates one input or hundreds from the same partition. Mutation testing, in contrast, reflects the ability of a test suite to expose behavioral differences within those paths, making it a suitable metric for evaluating which generalized tests provide additional fault detection capability [40].

Mutation testing systematically introduces small faults into program code and measures whether test suites detect them. The approach rests on two hypotheses: the competent programmer hypothesis (real faults are small deviations from correct programs) and the coupling effect (tests that detect simple faults also detect more complex ones) [60]. These hypotheses justify using small syntactic changes as proxies for real programming errors. Mutation operators alter program statements to create mutants. Common operators include arithmetic replacements (e.g., `+` to `-`), relational boundary shifts (e.g., `>` to `>=`), logical connector changes (e.g., `&&` to `||`), constant modifications, and replacements of return values with defaults such as `0`, `true`, or `null` [40]. A test suite's mutation score, i.e., the proportion of mutants it detects ("kills"), has been shown to correlate with real fault detection capability [42, 62].

The limitations of single-input tests become clear through mutation analysis. A unit test verifying $abs(0) = 0$ cannot kill a mutant changing `x >= 0` to `x == 0`, because the test's single input still satisfies the mutated condition. A property-based test that exercises the same execution path with multiple inputs will detect this mutant when positive values produce negative results. This difference reveals both the improvement potential of test generalization and provides a concrete criterion for selecting which generalized tests to retain. Only those generalizations that detect mutants not caught by existing tests contribute unique fault detection capability to the test suite, whereas generalizations that do not kill any new mutants only increase test suite size and runtime without any tangible benefits.

3 Approach

This section presents our semantics-based approach for automated test generalization. We implemented this approach in Teralizer, a prototype tool for Java. As shown in Figure 3, our approach follows a five-stage pipeline that takes the implementation and test code of a software project as input and produces generalized property-based tests as output: (1) test and assertion analysis identifies potentially generalizable tests and their assertions, (2) tested method identification maps assertions to the methods that they validate through data flow analysis, (3) specification extraction recovers input/output specifications from the tested methods through single-path symbolic analysis, (4) generalized test creation produces property-based tests with three input generation strategies (Baseline, Naive, and Improved), and (5) test suite reduction filters tests to retain only those that measurably improve the mutation score of the test suite.

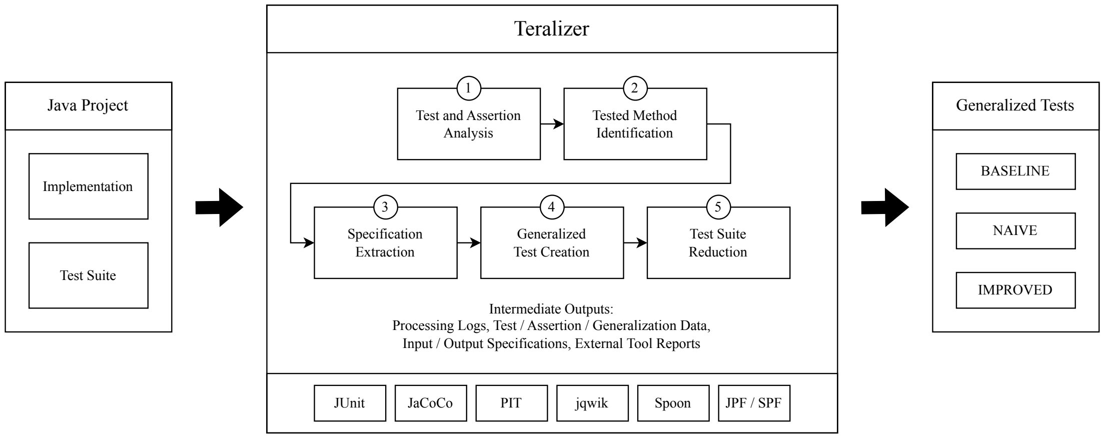

Java Project
Implementation
Test Suite
Teralizer
1
Test and Assertion Analysis
2
Tested Method Identification
3
Specification Extraction
4
Generalized Test Creation
5
Test Suite Reduction
Intermediate Outputs:
Processing Logs, Test / Assertion / Generalization Data,
Input / Output Specifications, External Tool Reports
JUnit
JaCoCo
PIT
jqwik
Spoon
JPF / SPF
Generalized Tests
BASELINE
NAIVE
IMPROVED

Fig. 3. Overview of Teralizer's test generalization process.

We illustrate our approach with a running example. Listing 1 shows a bonus calculation method with three execution paths for exceptional, good, and bad performance. Listing 2 shows a typical unit test that tests one input for each performance level. While the test detects regressions that affect these three inputs, it misses regressions that affect other inputs in the same partitions. Consider a mutation that changes `sales / 2 >= target` to `sales / 2 > target`. Despite the change, the test still passes, but boundary cases where `sales = 2 * target` now return the good performance bonus instead of the exceptional performance bonus. To detect such subtle regressions, Teralizer transforms existing unit tests into property-based tests that encode the intended behavior for all inputs in the covered input partitions.

While the underlying generalization approach is independent of specific programming languages or project environments, our current implementation of Teralizer targets Java 5–8 projects (imposed by our dependency on Symbolic PathFinder) that use Maven or Gradle for dependency management and JUnit 4 or JUnit 5 for testing. Before starting the main processing stages, Teralizer automatically detects the build system used by the target project and injects necessary dependencies including jqwik [41] for property-based testing, PIT [18] for mutation testing, and JaCoCo [54] for coverage tracking, thus ensuring full automation without the need for manual preprocessing steps.

The following subsections detail each stage of the generalization process as implemented in Teralizer. Section 3.1 explains test and assertion analysis. Section 3.2 describes tested method identification. Section 3.3 presents input/output

Listing 1. Implementation of the `calculate` method.

```java
class BonusCalculator {
  int calculate(int sales, int target) {
    if (sales / 2 >= target) {
      // exceptional performance
      return sales / 10;
    } else if (sales >= target) {
      // good performance
      return sales / 20;
    }
    // bad performance
    return 0;
  }
}
```

Listing 2. Original test for the `calculate` method.

```java
@Test
void testCalculate() {
  BonusCalculator c = new BonusCalculator();
  // exceptional performance:
  int b1 = c.calculate(2500, 1000);
  // good performance:
  int b2 = c.calculate(1500, 1000);
  // bad performance:
  int b3 = c.calculate(500, 1000);
  assertEquals(250, b1);
  assertEquals(75, b2);
  assertEquals(0, b3);
}
```

specification extraction, and Section 3.4 covers generalized test creation. Finally, Section 3.5 describes mutation-based test suite reduction. The full implementation of Teralizer is available in our replication package [32].

3.1 Test and Assertion Analysis

To identify generalization candidates, Teralizer collects descriptions of all tests and assertions in the codebase. First, it executes the original test suite to generate JUnit XML reports. Next, it parses these reports to identify executed tests and their execution results. For every test in the reports, Teralizer conducts static analysis via Spoon [65] to extract source code locations, test annotations, used assertions, involved data types, and various other structural information relevant for the generalization (full data available in our replication package [32]). Based on the collected information, Teralizer applies filtering heuristics to exclude unsuitable tests and assertions from further processing.

Tests need to pass three filters: `TestType`, `NonPassingTest`, and `NoAssertions`. The `TestType` filter rejects tests that are not standard `@Test` methods, such as `@ParameterizedTests` that would require special handling not currently implemented in Teralizer, or `@RepeatedTests` that indicate non-deterministic behavior incompatible with symbolic analysis-based specification extraction (as described in Section 2.3). The `NonPassingTest` and `NoAssertions` filters both address the absence of validated oracles. Failing tests do not reflect intended behavior, so no specification can be inferred from them. Similarly, tests without assertions lack explicit oracles. Assertions might be missing for two reasons: (i) the test validates only that execution completes without crashing or (ii) assertions are present but are contained in helper methods that are not captured by Teralizer's intraprocedural assertion detection. While interprocedural analysis could detect delegated assertions and test refactoring could make implicit validation explicit, we leave such enhancements for future work. Sections 4.6 and 4.7 quantify how frequently each filter excludes tests in practice.

Individual assertions within suitable tests require further analysis to determine generalizability. Teralizer currently supports four assertion types: `assertEquals`, `assertTrue`, `assertFalse`, and `assertThrows` from both JUnit 4 and JUnit 5. These assertions capture computational relationships that symbolic analysis can model. The `AssertionType` filter excludes unsupported assertions such as reference equality checks (`assertSame`, `assertNull`) and structural comparisons (`assertArrayEquals`, `assertInstanceOf`) that operate on data types current symbolic execution cannot accurately model. The `ExcludedTest` filter excludes assertions from tests already filtered at the test level.

3.2 Tested Method Identification

After identifying generalizable tests and assertions, the next step is to identify which implementation methods are validated by the tests. These methods under test (MUT) serve as targets for subsequent specification extraction. First, we must distinguish test setup code from code that exercises and validates the MUT to be able to restrict specification extraction to relevant parts of the implementation. Second, when a test validates multiple MUTs, we must determine which MUT call corresponds to each assertion so we can replace the expected value used in the assertion with the right output specification when creating the corresponding generalized test. Teralizer achieves this through static analysis based on Spoon [65] that traces output values validated by assertions back to the method calls that produced them.

Consider the `testCalculate` method in Listing 2. To identify MUT calls in this method, Teralizer uses Spoon to first get the `actual` arguments that are passed into each assertion. In our example, all three assertions are `static void assertEquals(int expected, int actual)` calls. Teralizer thus identifies `b1`, `b2`, and `b3` as the `actual` arguments of the three assertions. Since all three arguments are local variables, Teralizer uses Spoon to identify where they were defined. For example, for `b1`, Spoon identifies `int b1 = c.calculate(2500, 1000)` as the definition. Since the right side of the assignment is a method call, Teralizer marks it as a MUT call, storing a description of the method as well as a mapping to the corresponding `assertEquals(250, b1)` call. Processing `b2` and `b3` similarly identifies `c.calculate(1500, 1000)` and `c.calculate(500, 1000)` as the MUT calls for the second and third assertions.

Processing for other assertion types and code structures follows similar patterns. Teralizer first identifies the assertion argument that represents the (output of the) MUT call. If the argument is a method call, Teralizer directly marks it as a MUT call. Otherwise, Teralizer aims to identify a MUT call from the argument. For local variables, it traces them back to their definition using the simple data flow analysis based on Spoon we described in the previous paragraph. For lambda expressions, which are commonly used as the `executable` argument of `assertThrows(Class expectedType, Executable executable)` assertions, Teralizer instead marks the last method call within the lambda expression as the MUT call, following the heuristic that the last call typically triggers the expected exception.

Teralizer applies three filters to exclude unsuitable MUTs: `MissingValue`, `ParameterType`, and `ReturnType`. The `MissingValue` filter rejects cases where Teralizer cannot identify a MUT for an assertion or cannot extract a method signature for an identified MUT. Common causes for this include reversed `expected` and `actual` arguments (e.g., `assertEquals(abs(0), 0)`), validation of object fields set as side effects rather than return values (e.g., `assertEquals(3, a.length)`), and MUTs in inheritance hierarchies that Spoon cannot resolve. The `ParameterType` and `ReturnType` filters reject MUTs that use unsupported types for all of the MUT's parameters or return values. While methods with mixed parameter types can be partially generalized (numeric and boolean parameters become test inputs; others remain unchanged), SPF cannot extract complete constraints for strings, arrays, and objects. This is due to symbolic analysis limitations discussed in Section 2.3. Sections 4.6 and 4.7 evaluate the exclusion rates of all filters.

3.3 Specification Extraction

The specification extraction stage takes the MUT-to-assertion mappings from tested method identification and produces input/output specifications for every MUT. Each specification captures two elements: the path condition that describes which inputs follow the same execution path through the MUT as the test, and the symbolic output expression that describes expected results for any input satisfying the path condition. Teralizer extracts specifications through a two-step process: first instrumenting tests to create entry points for symbolic analysis, then executing them with Symbolic PathFinder (SPF) in constraint collection mode. In this mode, SPF follows the test's execution path while maintaining symbolic representations, extracting path-exact specifications without exploring alternative paths.

The first step, test instrumentation, generates three artifacts for each identified MUT: an instrumented version of the test class, a driver class, and a configuration file for SPF. In our running example, the `Instrumented` test class for the *good performance* MUT call (Listing 3) wraps the `c.calculate(1500, 1000)` call in a `wrapper` method that marks the starting point for symbolic analysis. The `Driver` class (Listing 3) provides the entry point for SPF. It instantiates the instrumented test class, runs setup code in methods annotated with `@Before`, and executes the targeted test method `testCalculate`. The SPF configuration (Listing 4) sets up symbolic analysis of the `wrapper` method, registers a custom `TestGeneralizationListener` for specification extraction, and configures relevant resource limits.

The second step, symbolic analysis, executes these artifacts with SPF. For the *good performance* case with concrete inputs (1500, 1000), the first if condition `sales / 2 >= target` in the `calculate` method (see Listing 1) evaluates to false, so SPF records the negated constraint `sales / 2 < target`. The second if condition `sales >= target` evaluates to true, adding `sales >= target` to the accumulated path condition. When the wrapper method returns, our custom `TestGeneralizationListener` captures the complete path condition (`sales / 2 < target && sales >= target`) and the symbolic output expression (`sales / 20`), writes both concrete input/output values and symbolic input/output specifications to JSON files, and then immediately terminates SPF without exploring alternative paths. Listing 5 shows the input/output specifications that are collected for the three identified MUT calls of our running example.

Single-path symbolic analysis requires tested methods to be pure functions, i.e., deterministic, side-effect-free, and dependent only on their input parameters [5, 12]. Furthermore, SPF can only provide precise specifications for numeric and boolean values. Because of this, Teralizer only targets generalization of numeric and boolean inputs, leaving string, array, and object inputs unchanged. If no input/output specification can be extracted for a given MUT, Teralizer excludes this MUT from further processing. The primary causes of such exclusions are SPF errors, NullPointerExceptions for certain edge cases in our current implementation of Teralizer, and exceeded resource limits. By default, Teralizer uses a 60 second timeout per MUT, a 100,000-character limit per path condition, and a function call depth limit of 100 (Listing 4). We empirically determined these settings to provide a reasonable trade-off between resource consumption and result quality. Sections 4.6 and 4.7 show how often the mentioned causes lead to exclusions in our evaluation.

3.4 Generalized Test Creation

The generalized test creation stage transforms original JUnit tests into property-based jqwik tests. Figure 2 provides a high-level overview of this transformation: the original test with hardcoded values (left) becomes a property-based test (right) where inputs are automatically generated to satisfy constraints and expected values are encoded as expressions that hold for all inputs from the input partition. This preserves the developer-provided oracles encoded in assertions while generalizing from concrete values to symbolic specifications. To be able to systematically evaluate the costs and benefits of test generalization, Teralizer creates three variants of each property-based test. The Baseline variant tests only the original inputs to measure framework overhead. The Naive variant adds random input generation to explore the input space, and the Improved variant incorporates constraint-aware generation to favor values at the boundaries

Listing 3. Driver and instrumented test class used for specification extraction of the *good performance* case.

```java
public class Driver {
  // Driver.main provides entry point for SPF:
  public static void main(String[] args) {
    Instrumented i = new Instrumented();
    i.testCalculate();
  }
}

public class Instrumented {
  @Test
  void testCalculate() {
    BonusCalculator c = new BonusCalculator();
    ...
    // Instrumented.wrapper marks the
    // starting point for symbolic analysis:
    int b2 = this.wrapper(c, 1500, 1000);
    ...
  }
  int wrapper(
    BonusCalculator c, int sales, int target
  ) {
    return c.calculate(sales, target);
  }
}
```

Listing 4. Symbolic PathFinder configuration used for specification extraction of the *good performance* case.

```
target=Driver
symbolic.method=Instrumented.wrapper(con#sym#sym)
symbolic.collect_constraints=true

listener=teralizer.jpf.TestGeneralizationListener

teralizer.max_execution_time=60.0
teralizer.max_path_condition_size=100000
search.depth_limit=100
...
```

Listing 5. Input/Output specifications extracted for the MUT calls in the testCalculate method.

```
exceptional performance:
- input:  sales / 2 >= target
- output: sales / 10

good performance:
- input:  sales / 2 < target && sales >= target
- output: sales / 20

bad performance:
- input:  sales / 2 < target && sales < target
- output: 0
```

of input partitions. Section 3.4.1 describes the main transformation process, Section 3.4.2 explains differences between the three variants, and Section 3.4.3 provides further details on the constraint encoding strategy used by Improved.

3.4.1 *Transformation Process.*

For each generalizable assertion with a successfully extracted input/output specification, Teralizer creates a new test class containing a single property-based test. Listing 6 shows the result of this transformation for our running example's *good performance* case. Compared to the original test shown in Listing 2, the new property-based test (i) replaces the `@Test` annotation with `@Property` to specify an input supplier and execution count, (ii) adds a `TestParams` parameter to receive generated inputs, (iii) substitutes hardcoded MUT arguments in `calculate(1500, 1000)` to produce the new MUT call `calculate(_p_.sales, _p_.target)`, and (iv) replaces the concrete expected value `75` with the oracle call `calculateExpected(_p_)` which calculates expected outputs from the extracted specification (see Listing 7). The `TestParams` class shown in Listing 9 encapsulates the generated sales and target values, while the `BaselineSupplier` in Listing 8 demonstrates how the Baseline variant provides input values for these parameters by wrapping the concrete values extracted during SPF execution of the original test.

Several factors influence the design of the created classes. Instead of modifying existing classes, Teralizer creates one new test class per assertion to isolate generalization effects. This is relevant for mutation testing: PIT requires a green test suite and only supports class-level exclusion, so a single failing generalized assertion could otherwise prevent all tests in the same class from being evaluated. Furthermore, the isolation prevents unintended side-effects on developer-written code. Input parameters are provided by supplier classes rather than parameter annotations to enable encoding of constraints that reference multiple parameters. Only the generalized assertion is preserved — other assertions are removed because they might validate methods that consume outputs from the generalized MUT and fail when those outputs differ from the original test's values. Non-generalizable parameters (strings, arrays, objects) retain their concrete input values. This enables partial generalization when some parameters cannot be generalized.

3.4.2 *Three-Variant Design.*

Teralizer creates three variants of each property-based test that differ only in their input generation strategies. This design allows us to isolate and measure distinct aspects of test generalization: pure framework overhead (Baseline), benefits from testing additional inputs (Naive), and improvements from constraint-aware input

Listing 6. Generalized test for the *good performance* case.

```java
@Property(
  supplier = BaseLineSupplier.class,
  tries = 200
)
void testCalculate(TestParams _p_) {
  BonusCalculator c = new BonusCalculator();
  // exceptional performance:
  int b1 = c.calculate(2500, 1000);
  // good performance:
  int b2 = c.calculate(_p_.sales, _p_.target);
  // bad performance:
  int b3 = c.calculate(500, 1000);
  assertEquals(calculateExpected(_p_), b2);
}
```

Listing 7. Output oracle for the *good performance* case.

```java
int calculateExpected(TestParams _p_) {
  return _p_.sales / 20;
}
```

Listing 8. Baseline supplier for *good performance* inputs. The supplier uses the same inputs as the original test.

```java
class BaselineSupplier {
  Arbitrary get() {
    return Arbitraries.just(
      new TestParams(1500, 1000));
  }
}
```

Listing 9. Container class for generated input values.

```java
class TestParams {
  int sales;
  int target;
  TestParams(int sales, int target) {
    this.sales = sales;
    this.target = target;
  }
}
```

Listing 10. Naive supplier for *good performance* inputs. The supplier generates random inputs and then filters them to only test values that match the input specification.

```
class NaiveSupplier {
  Arbitrary get() {
    return Arbitraries.integers().flatMap(
      sales -> Arbitraries.integers().map(
        target -> new TestParams(sales, target)))
    .filter(this::satisfiesInputSpec);
  }
}
```

Listing 11. Input filter for the *good performance* case.

```
boolean satisfiesInputSpec(TestParams _p_) {
  return _p_.sales / 2 < _p_.target
    && _p_.sales >= _p_.target;
}
```

Listing 12. Improved supplier for *good performance* inputs. The supplier partially encodes the input specification during generation, reducing filtering failures compared to Naive.

```
class ImprovedSupplier {
  Arbitrary get() {
    return Arbitraries.integers().flatMap(
      target -> Arbitraries.integers()
        // sales >= target is encoded
        // sales / 2 < target is not encoded
        .between(target, Integer.MAX_VALUE)
        .map(sales ->
          new TestParams(sales, target)))
    .filter(this::satisfiesInputSpec);
  }
}
```

generation (Improved). To reduce evaluation costs, Teralizer shares common analysis stages across all three variants. Test and assertion analysis, tested method identification, and specification extraction execute only once per test, with their results reused for each generalization strategy. This architecture ensures fair comparison while avoiding redundant computation — all three generalization strategies work from identical specifications and analysis results, eliminating confounding factors such as variation in analysis time or non-deterministic failures from `OutOfMemoryErrors`.

The **Baseline** variant uses only the original test's input values to isolate the overhead of the property-based testing framework. As shown in Listing 8 for the *good performance* case, the supplier returns `Arbitraries.just(new TestParams(1500, 1000))`, providing exactly the same inputs as the original test. This enables us to quantify the infrastructure cost of property-based testing, i.e., test orchestration, parameter injection, and jqwik's execution machinery, without the cost of additional input generation and repeated test execution. By establishing this baseline overhead, we can isolate the cost of input generation in Naive and Improved. Any runtime beyond Baseline is attributable to generation and filtering strategies rather than basic property-based testing infrastructure.

The **Naive** variant adds random input generation combined with post-generation filtering to demonstrate the benefits of testing additional inputs beyond the original ones. As shown in Listing 10, Teralizer uses jqwik's `Arbitraries` classes to generate random integer values for `sales` and `target`, then applies the `satisfiesInputSpec` filter shown in Listing 11 to retain only values that match the constraints encoded in the extracted input specification `sales / 2 < target && sales >= target` which we previously showed in Listing 5. To ensure that created property-based tests always cover at least the same mutants as the corresponding original tests, all Naive and Improved suppliers contain additional code that always selects the original inputs as the first set of inputs exercised by the property-based tests. The implementation of this logic is straightforward, but is excluded from the listings for brevity.

The **Improved** variant implements a constraint-aware input generation strategy to address the limitations of purely random input selection in the presence of restrictive input constraints. For example, `a == b && b == c` is unlikely to be satisfied by randomly assigned `a`, `b`, and `c`, causing jqwik to throw a `TooManyFilterMissesException` after too many failed input generation attempts. This, in turn, prompts Teralizer to exclude the property-based test from the test suite. To address this limitation, the Improved variant encodes some constraints directly in the input supplier. For example, as Listing 12 shows for the *good performance* case, the generated supplier enforces the constraint `sales >= target` via the call `between(target, Integer.MAX_VALUE)`. This partial encoding of constraints increases the likelihood that a valid input is selected before filtering by reducing the size of the input space from which values are chosen.

3.4.3 Constraint-Aware Generation. The constraint-aware input generation strategy used by the Improved variant encodes simple equality and inequality constraints such as `x == y`, `x < 10`, or `y >= x` where both sides of the (in-)equality are either variables or constants. More complex constraints are not encoded in the initial input value generation but are enforced during filtering. For example, the input constraint `sales / 2 < target` is not represented in the input generation code of the `ImprovedSupplier` shown in Listing 12. However, generated inputs that violate this constraint are still rejected by the `filter(this::satisfiesInputSpec)` call. This ensures that even if input constraints can only be partially encoded, all exercised input values are guaranteed to satisfy the complete input specification.

Algorithm 1 shows the input generation logic. It first assigns indices based on parameter order (line 1), then processes each encodable constraint (lines 2-10) to handle circular dependencies: constraints are only added to the parameter with the higher index, with constraint directions rewritten as needed. This ensures each parameter depends only on previously generated ones. For example, given `a >= b && b >= a` with `a` at index 0 and `b` at index 1, we add `b <= a` (rewritten from `a >= b`) and `b >= a` to parameter `b`, while `a` remains unconstrained. During generation (lines 11-24), the

Algorithm 1 Constraint-Aware Input Generation (Improved)

```
Require: Input specification $S$, Parameters $P = \{p_1, \ldots, p_n\}$
Ensure: Generated test inputs satisfying $S$
 1: Assign indices: $idx(p_i) = i$ for $i \in \{1, \ldots, n\}$
 2: for all constraints $c \in S$ of form $p_i \odot p_j$ where $\odot \in \{=, <, \leq, >, \geq\}$ do
 3:     if $idx(p_i) > idx(p_j)$ then
 4:         Add constraint to $p_i$ based on $p_j$
 5:     end if
 6:     if $idx(p_j) > idx(p_i)$ then
 7:         Rewrite constraint and add to $p_j$ based on $p_i$
 8:         E.g., $p_i < p_j$ becomes $p_j > p_i$
 9:     end if
10: end for
11: for each parameter $p_i$ in index order do
12:     $E_i \leftarrow$ equality constraints for $p_i$
13:     $L_i \leftarrow$ lower bound constraints for $p_i$
14:     $U_i \leftarrow$ upper bound constraints for $p_i$
15:     if $E_i \neq \emptyset$ then
16:         Generate $p_i$ = value from first equality constraint
17:     else if $L_i \neq \emptyset$ or $U_i \neq \emptyset$ then
18:         $lower \leftarrow \max(L_i)$ if exists, else type minimum
19:         $upper \leftarrow \min(U_i)$ if exists, else type maximum
20:         Generate $p_i \in [lower, upper]$
21:     else
22:         Generate $p_i$ randomly within type bounds
23:     end if
24: end for
25: Apply filter for non-encodable constraints
26:
27: return generated inputs if filter passes
```

algorithm selects the strictest applicable bounds: equality constraints take precedence, followed by the highest lower bound and lowest upper bound. Thus, `a` generates freely, while `b` generates from interval `[a, a]`, effectively encoding `a == b`. Line 25 applies filtering for non-encodable constraints, ensuring all inputs satisfy the full specification.

While this partial constraint encoding cannot eliminate all `TooManyFilterMissesExceptions`, it reduces their prevalence by constraining the space from which inputs can be selected. Additionally, constraint-aware input generation enables jqwik's `Arbitraries` to more reliably produce inputs at the boundaries of input partitions. For example, consider `x >= 0 && x <= 1000`. In the Naive variant, jqwik has no knowledge of the true partition boundaries. Thus, the used `Arbitraries` produce assumed boundary values such as `Integer.MIN_VALUE`, `Integer.MAX_VALUE - 1`, etc. While these are quickly excluded by filtering, coverage of the true boundary values such as 0, 1, 999, and 1000 is then left up to chance. In contrast, both boundaries of this example are exactly encoded in `Arbitraries` calls of the Improved variant, enabling them to reliably produce the true boundary values. We deliberately avoid full constraint solving for input generation because it would introduce significant runtime overhead and would still require fallbacks for cases that cannot be solved, either because they are not tractable for current solvers or because they are inherently undecidable.

3.5 Test Suite Reduction

Despite covering many more inputs than the original tests, some property-based tests do not detect any additional faults. Thus, these tests increase test suite size and runtime but do not provide any tangible benefits in exchange for this. Similarly, successful generalization may render some original tests redundant. This is because tests created by all three of Teralizer's generalization variants are designed to use the original input values as the first set of inputs exercised during property-based test execution. To address these inefficiencies, Teralizer performs test suite reduction as the final stage of the generalization pipeline, using mutation testing to measure each test's contribution to fault detection and retaining only those original and generalized tests that strengthen the test suite's effectiveness.

Teralizer evaluates fault detection capability using the `DEFAULTS` group of mutation operators provided by PIT [18]. This group provides a stable set of operators that minimize equivalent mutants and avoid subsumption [16, 17]. The full set of operators is listed in Table 1. Each row shows the name of the mutator, a short description of its behavior, and a source code representation of the mutator's effects. The operators can be roughly categorized by the type of mutation they produce. The first subgroup modifies arithmetic operations. The second one replaces return values. The third one modifies conditionals, and the fourth one removes calls to methods that have `void` as their return type.

To perform test suite reduction, Teralizer first executes mutation testing on the original test suite as well as the non-reduced test suites created by the three test generalization variants. By comparing which mutants each configuration detects, Teralizer identifies generalized tests that catch mutants not detected by the original test suite. The selection criterion is straightforward: retain only generalized tests that detect at least one mutant not caught by the original test suite. This ensures that every generalized test in the final test suite contributes unique fault detection capability, while those that only detect already-caught mutants are excluded as redundant.

Beyond filtering generalized tests, Teralizer identifies original tests that can be removed without loss of test suite effectiveness. An original test is removable if property-based tests were successfully created for all of its assertions. For tests containing a single assertion, generalization ensures that the property-based test validates that assertion across the entire input partition, making the original single-input validation redundant. For tests containing multiple assertions, removal requires that every assertion has been successfully transformed. If any assertion cannot be generalized (due to type limitations, failed MUT identification, or other filtering criteria) the original test must be retained to preserve that validation. The final test suite of each variant combines the retained generalized tests with the retained original tests.

Table 1. Mutation operators included in PIT's DEFAULTS group.

		Example	
Mutator	Description	Before	After
Math	Replaces arithmetic operations	`x + y`	`x - y`
Increments	Replaces increment/decrement	`i++`	`i--`
InvertNegs	Inverts negation of variables	`return -x`	`return x`
BooleanTrueReturnVals	Returns `true` for booleans	`return b`	`return true`
BooleanFalseReturnVals	Returns `false` for booleans	`return b`	`return false`
PrimitiveReturns	Returns `0` for numeric primitives	`return a`	`return 0`
EmptyObjectReturnVals	Returns empty for strings	`return s`	`return ""`
NullReturnVals	Returns `null` for objects	`return o`	`return null`
RemoveConditionalEqualElse	Forces else for equality checks	`if (a == b)`	`if (false)`
RemoveConditionalOrderElse	Forces else for inequality checks	`if (a < b)`	`if (false)`
ConditionalsBoundary	Changes boundary of inequalities	`if (a < b)`	`if (a <= b)`
VoidMethodCall	Removes void method calls	`foo(...)`	`/* removed */`

4 Evaluation

We evaluate the benefits, costs, and limitations of semantics-based test generalization through six research questions:

- **RQ1:** How much does test generalization improve the mutation score of existing unit test suites?
- **RQ2:** How does constraint complexity affect constraint-aware versus random input generation?
- **RQ3:** To which degree does generalization affect the size and runtime of the target test suites?
- **RQ4:** How efficient is test generalization compared to test generation?
- **RQ5:** What are the causes of unsuccessful generalization attempts under controlled conditions?
- **RQ6:** What are the causes of unsuccessful generalization attempts under real-world conditions?

Section 4.1 describes our experimental setup and methodology. Sections 4.2–4.7 present results. All experiments were run on a MacBook Air (M2, 24 GB RAM) with default JVM settings. All data is available in our replication package [32].

4.1 Experimental Setup

To identify current capabilities and limitations of semantics-based test generalization, we systematically evaluate our implementation of Teralizer across a multitude of projects which range from controlled benchmark cases that are well-suited for test generalization to real-world projects that demonstrate which future advances are needed to improve practical applicability of test generalization tools. In this section, we describe key components of our experimental setup and establish a shared vocabulary that we use throughout the evaluation to refer to different stages of the processing pipeline, different test suite variants, and different groups of projects that are part of our evaluation dataset.

4.1.1 Processing Stages and Test Suite Variants. As shown in Figure 3, Teralizer's processing pipeline consists of five stages. The first three (test and assertion analysis, tested method identification, and specification extraction) are Shared stages that are only executed once per pipeline run because their results can be reused across generalization strategies (Baseline, Naive, and Improved). Processing starts with all Original tests. However, each stage can exclude tests that are unsuitable for further processing. We refer to the subset of Original tests that remain after the Shared processing stages as the Initial test suite. The last two processing stages (generalized test creation and test suite reduction) are executed once for the Baseline generalization strategy and three times each for the Naive and Improved strategies using different values for jqwik's `tries` setting. Thus, we distinguish the following nine test suite variants:

- Original: before any processing has taken place,
- Initial: after exclusions by Shared processing stages,
- Baseline: after Baseline generalization and reduction,
- Naive_{10}, Naive_{50}, Naive_{200}: after Naive generalization and reduction (10/50/200 `tries`),
- Improved_{10}, Improved_{50}, Improved_{200}: after Improved generalization and reduction (10/50/200 `tries`).

As described in Section 3.4.2, using Initial as a shared starting point not only reduces the runtime costs of the evaluation but also enables a fair comparison across strategies by avoiding non-deterministic Stage 1–3 failures such as `OutOfMemoryErrors` from affecting one strategy more strongly than another. The used `tries` settings of 10, 50, and 200 were selected to demonstrate the scaling behavior of higher `tries` while keeping runtime costs manageable.

4.1.2 Evaluated Projects. Our evaluation employs three complementary datasets that progressively reveal the gap between controlled and real-world conditions for test generalization. The EqBench benchmark [4] provides numeric-focused programs that are well-suited for symbolic analysis. Utility methods extracted from Apache Commons projects bridge toward real-world complexity while maintaining the focus on numeric constraints. Projects from the RepoReapers dataset [55] expose real-world applicability challenges. Table 2 provides descriptive statistics of the datasets. For the implementation, we show the number of files, classes, and source lines of code (SLOC). For tests, we additionally provide the number of test methods, i.e., methods that are annotated with `@Test`, `@RepeatedTest`, or `@ParameterizedTest`.

EqBench. The EqBench benchmark [4] (rows eqbench-es-* in Table 2) provides controlled conditions for automated test generalization. Originally designed for equivalence checking research, its 652 Java classes implement equivalent and non-equivalent program pairs focusing on numeric computations while deliberately avoiding features that complicate automated reasoning (e.g., recursion, reflection, and complex object graphs). Since EqBench provides only implementation code without tests, we generated test suites using EvoSuite [29] with three different search budgets (1s, 10s, and 60s per implementation class), creating the dataset variants eqbench-es-1s, eqbench-es-10s, and eqbench-es-60s. This design explores how initial test suite quality affects generalization effectiveness: stronger initial suites offer better test diversity but less improvement potential due to their higher initial mutation scores.

Apache Commons. To bridge toward real-world complexity, we extracted numeric utility methods from Apache Commons projects (rows commons-utils-* in Table 2). Using Sourcegraph's code search [74], we identified public static methods with numeric or boolean parameters and return values — the types currently supported by Teralizer (search queries are available in our replication package [32]). This yielded 247 classes from 17 Apache Commons projects (commons-math, commons-numbers, commons-lang, etc.), including all transitively called methods and dependencies to ensure compilation (19,709 LOC total). From this, we created four dataset variants: commons-utils-es-1s, commons-utils-es-10s, and commons-utils-es-60s use EvoSuite-generated tests with 1s, 10s, and 60s per-class search budgets. In contrast, commons-utils-dev preserves the 725 original developer-written tests (14,389 LOC). This enables direct comparison of generalization effectiveness between developer-written tests and tests generated by EvoSuite.

RepoReapers. To understand current limitations in practical settings, we selected 632 projects from RepoReapers [55], a curated collection of 1.9 million GitHub repositories specifically filtered for their use of sound software engineering practices (e.g., extensive development history, use of software testing and issue tracking, availability of documentation). Our selection criteria balanced technical constraints with evaluation goals. All selected projects target Java 5–8 (for SPF compatibility), use JUnit 4 or 5 through Maven (for Teralizer compatibility), have standard directory structures

Table 2. Number of files, classes, source lines of code (SLOC), and test methods per project.

	Implementation			Test			
Project	Files	Classes	SLOC	Files	Classes	SLOC	Methods
eqbench-es-1s	544	652	27,871	544	544	35,666	4,718
eqbench-es-10s	544	652	27,871	543	543	36,937	4,875
eqbench-es-60s	544	652	27,871	544	544	37,836	4,974
commons-utils-es-1s	106	247	19,709	103	103	17,524	2,481
commons-utils-es-10s	106	247	19,709	103	103	19,082	2,738
commons-utils-es-60s	106	247	19,709	102	102	18,839	2,735
commons-utils-dev	106	247	19,709	80	119	14,389	725
repo-reapers (total)	41,292	50,474	2,735,127	22,281	30,894	2,012,601	81,810
repo-reapers (mean)	65	79	4,320	35	48	3,179	162
repo-reapers (median)	49	56	3,253	23	26	2,107	86

(for automated processing), medium-sized codebases (5,000–50,000 LOC), and substantial test suites (20–80% of total code). The selected projects collectively comprise 50,474 implementation classes and 30,894 test classes across diverse domains and coding styles. While Teralizer succeeds on EqBench and partially on Apache Commons (RQ1–RQ5, Sections 4.2–4.6), the RepoReapers projects expose current barriers to practical applicability (RQ6, Section 4.7).

4.2 RQ1: How much does test generalization improve the mutation score of existing unit test suites?

Mutation testing provides a rigorous effectiveness measure by evaluating a test suite's ability to detect deliberately introduced faults (Section 2.4). For Teralizer, it reveals whether testing additional inputs for existing execution paths achieves the intended improvement in fault detection capabilities. We use mutation score rather than parameter value coverage [71] — the metric employed by JARVIS [66] — because mutation testing is a stronger proxy for fault detection capability. Direct comparison with JARVIS is not possible as its implementation is not publicly available. Section 4.2.1 quantifies detection improvements across projects, and Section 4.2.2 analyzes improvements by mutation operator.

4.2.1 Overall Mutation Detection Rates. Generalization improves mutation detection across all eqbench-es-∗ and commons-utils-∗ projects, though the degree of improvement varies by project type (Figure 4). eqbench-es-∗ projects (rows 1–3 in the figure) show the largest improvements: detection rates increase from 48.1–51.6% to 49.5–55.0% across different `tries` and generalization strategies, representing absolute improvements of 1.2–3.9 percentage points (2.4–8.2% relative increase). commons-utils-es-∗ projects (rows 4–6) improve by 0.82–1.33 percentage points (1.4–2.3% relative increase), while commons-utils-dev (row 7) improves by only 0.05–0.07 percentage points from its 80.4% baseline.

Two factors strongly affect generalization improvements: initial test suite strength and input constraint complexity. eqbench-es-∗ projects offer the most suitable conditions for generalization. This is because EvoSuite-generated tests leave more room for enhancement (starting from 48.1–51.6% detection rates) than the thorough commons-utils-dev test suite, and the EqBench benchmark's input constraints are simpler than the input constraints of commons-utils-∗, thus making valid input generation easier (as discussed in more detail in Section 4.3). commons-utils-es-∗ projects face one additional challenge: while they also start from EvoSuite-generated tests, the commons-utils-∗ methods involve more complex input specifications, making it harder to generate inputs that satisfy these input constraints. commons-utils-dev

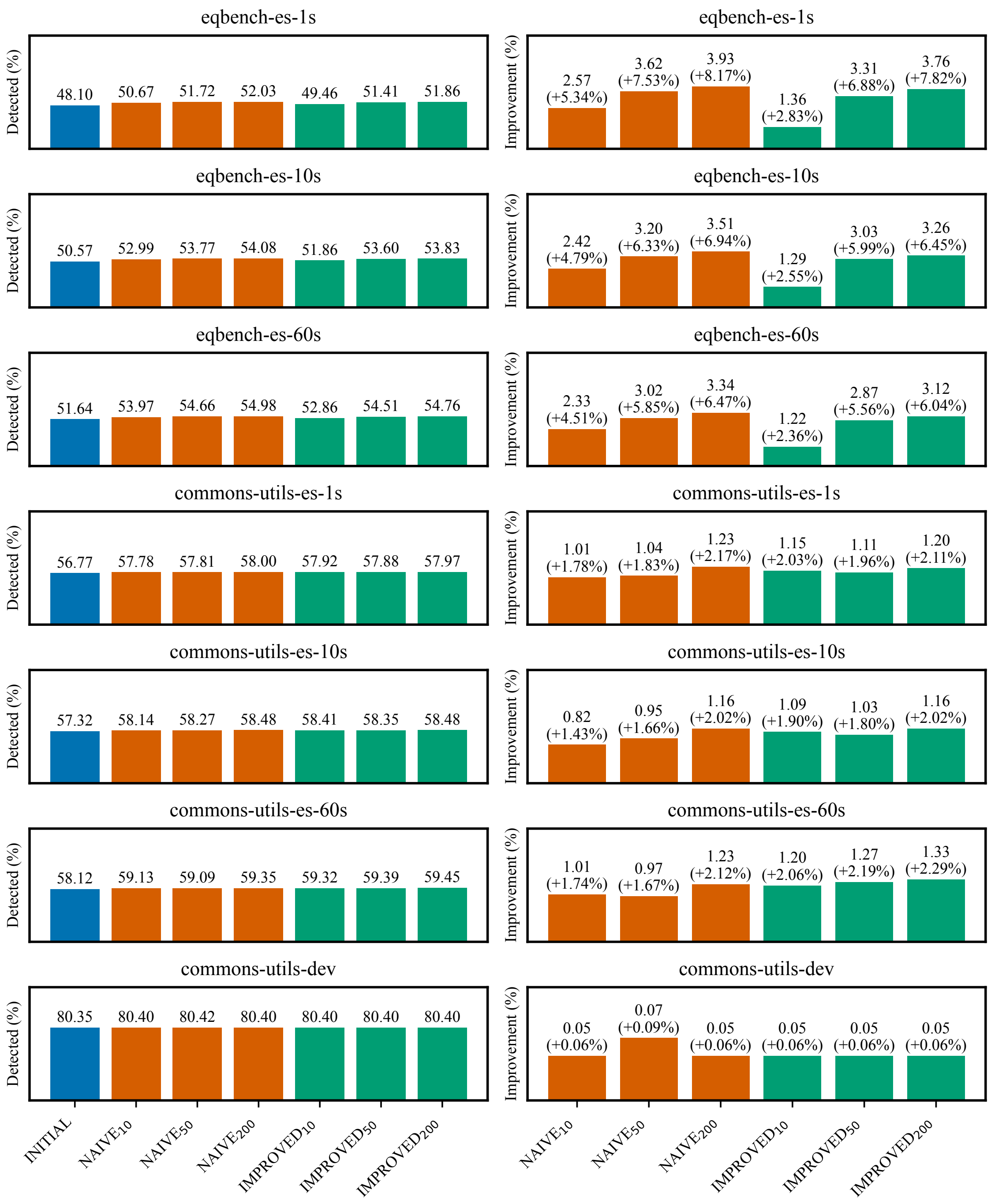


Fig. 4. Percentage of detected mutants (left side) and improvement over INITIAL (right side) per project and generalization strategy. Improvements show both the absolute improvement (top value) as well as the relative improvement (bottom value).

faces both challenges: the mature developer-written tests already achieve 80.4% detection and share the same complex constraints as other commons-utils-∗ variants. This leaves little opportunity for automated improvement.

Comparing the effectiveness of Naive and Improved shows opposite results across eqbench-es-∗ and commons-utils-es-∗ projects. On eqbench-es-∗ projects (rows 1–3), Naive (orange bars) outperforms Improved (green bars) for all `tries` settings, with gaps ranging from 0.17–1.21 percentage points. This is most pronounced with limited `tries`: Naive$_{10}$ achieves 50.67–53.97% detection rate while Improved$_{10}$ reaches only 49.46–52.86%. In contrast, commons-utils-es-∗ (rows 4–6) shows Improved outperforming Naive in 7 of 9 comparisons. These results reflect the following patterns: Naive performs better on `Math` mutations (59.1% of all mutants) while Improved better detects most other mutations and more effectively handles more complex constraints. In eqbench-es-∗, the high prevalence of `Math` mutations and low constraint complexity favor Naive, while in commons-utils-es-∗, Improved's benefits overcome its `Math` detection disadvantage. We investigate the mechanisms behind these results in further detail in Sections 4.2.2 and 4.3.

Higher `tries` settings improve detection rates, albeit with diminishing returns. One notable exception to this pattern appears in Improved variants with only 10 `tries`: on eqbench-es-∗ projects (rows 1–3), Improved$_{10}$ achieves only 2.4–2.8% relative improvement versus 5.6–6.9% for Improved$_{50}$ and 6.0–7.8% for Improved$_{200}$. This comparatively low increase in detection rates stems from boundary-focused generation consuming most of the limited `tries` of this variant, leaving insufficient attempts for testing intermediate values. With more `tries`, Improved variants perform comparably to or better than Naive variants as enough attempts remain for both boundary and non-boundary testing. commons-utils-es-∗ projects (rows 4–6) show less pronounced degradation at low `tries`, because their more complex constraints (Section 4.3) cause more boundary inputs to fail filtering, forcing earlier exploration of non-boundary values.

Combining short test generation with subsequent generalization can outperform longer test generation alone. On eqbench-es-∗ projects, 1-second EvoSuite generation followed by Naive$_{200}$ generalization achieves 52.0% mutation detection rate (last orange bar in row 1), surpassing 60-second generation alone (51.6%, blue bar in row 3). Similarly, on commons-utils-es-∗, 10-second generation plus Naive$_{200}$ generalization reaches 58.5% (last orange bar in row 5) versus 58.1% for 60-second generation (blue bar in row 6). Section 4.5.2 investigates the efficiency trade-offs between different combinations of EvoSuite timeouts and Teralizer `tries` in further detail through Pareto front analysis.

4.2.2 Mutation Detection Rates per Mutator. Generalization effectiveness varies significantly across mutation operators (Table 3). The `Math` mutation operator dominates the mutation landscape in eqbench-es-∗ and commons-utils-∗ projects, making up 59.1% of total mutants. This strong representation of `Math` mutations is primarily due to the focus on numeric computations in these datasets. The next most commonly occurring mutations are `ConditionalsBoundary` and `RemoveConditionalOrderElse` mutations, each of which accounts for 10.99% of total mutants (both operators are applied at the same source code locations). In contrast, the least common operators, i.e., `Increments` (0.52%), `Boolean-FalseReturnVals` (0.24%), and `EmptyObjectReturnVals` (0.13%), together account for less than 1% of all mutants. This large difference in mutant prevalence means that improvements to `Math` detection rates have proportionally larger impact on overall mutation scores than improvements to less commonly occurring mutation types.

Initial detection rates show a clear pattern: return value mutants (i.e., `PrimitiveReturns`, `BooleanTrueReturn-Vals`, `BooleanFalseReturnVals`, and `EmptyObjectReturnVals`) are killed in a large majority of cases (87.87–98.77% detection), while behavioral mutants are more elusive. For example, `VoidMethodCall` mutations achieve a detection rate of only 24.96% because removing void method calls typically affects only internal state or produces side effects that are rarely verified by existing assertions. Because these cases often require new assertions to be added to achieve detection rate improvements, they are largely beyond the intended capabilities of Teralizer. In contrast, Initial

Table 3. Number of mutants and percentage of detections per mutator in eqbench-es-∗ and commons-utils-∗ projects.

					Detected %				
Mutator	Total	Total %	Min %	Max %	Initial	Naive$_{200}$		Improved$_{200}$	
Math	61,841	59.10	52.34	62.16	50.99	54.98	(+3.99)	54.36	(+3.37)
ConditionalsBoundary	11,501	10.99	8.50	11.94	27.68	28.89	(+1.21)	30.23	(+2.55)
RemoveConditionalOrderElse	11,501	10.99	8.50	11.94	61.08	62.29	(+1.21)	62.47	(+1.39)
PrimitiveReturns	7,731	7.39	6.15	10.09	89.42	89.63	(+0.20)	89.90	(+0.47)
RemoveConditionalEqualElse	5,536	5.29	3.21	10.40	58.80	60.87	(+2.07)	61.00	(+2.20)
InvertNegs	3,122	2.98	2.94	3.12	58.91	60.61	(+1.70)	60.99	(+2.08)
VoidMethodCall	973	0.93	0.58	1.35	24.96	24.96	–	25.49	(+0.53)
NullReturnVals	933	0.89	2.13	3.38	98.77	98.77	–	98.77	–
BooleanTrueReturnVals	569	0.54	0.17	1.44	98.55	98.55	–	98.55	–
Increments	546	0.52	0.50	0.54	72.81	73.38	(+0.57)	73.50	(+0.69)
BooleanFalseReturnVals	250	0.24	0.09	0.63	87.87	87.87	–	87.87	–
EmptyObjectReturnVals	141	0.13	0.41	0.43	90.30	90.30	–	90.30	–

detection rates of 27.68% for `ConditionalsBoundary` mutations, 61.08% for `RemoveConditionalOrderElse`, and 58.80% for `RemoveConditionalEqualElse` highlight opportunities for automated improvement. After all, all three mutations affect partition boundaries, which is where Improved generalization aims to generate additional test inputs.

Naive$_{200}$ achieves the largest improvements on `Math` (+3.99%), `RemoveConditionalEqualElse` (+2.07%), and `Invert-Negs` (+1.70%). `RemoveConditionalEqualElse` detection improves because the created property-based tests cover diverse inputs that trigger both branches of equality checks. `Math` detection similarly benefits from diverse inputs because arithmetic operations often produce different results across input ranges. In contrast, 4 of 5 return value mutators show no improvement: `NullReturnVals`, `BooleanTrueReturnVals`, `BooleanFalseReturnVals`, and `EmptyObjectReturnVals`. These already achieve high Initial detection rates (87.87–98.77%) because return value mutations often directly violate test assertions, leaving little room for improvement. Increasing detection rates beyond this high starting point would likely require additional assertions to be introduced, rather than test inputs to be varied.

Improved$_{200}$ demonstrates different strengths than Naive through constraint-aware input generation. Comparing the two variants across all 12 mutation operators: Improved$_{200}$ outperforms Naive$_{200}$ for 7 operators, achieves the same detection rate for 4 operators (the zero-improvement return value mutations), and underperforms for only 1 operator (`Math`). The largest advantage is observed for `ConditionalsBoundary` detection. Here, Improved$_{200}$ achieves a 2.55% detection rate improvement compared to Naive$_{200}$'s 1.21%, which confirms that constraint-aware input generation achieves its intended purpose. Furthermore, the slightly higher detection rates for several other mutators suggest that testing at partition boundaries may also benefit some mutators that do not directly modify boundary constraints.

The `Math` mutation results highlight the trade-off in boundary versus non-boundary testing: Improved$_{200}$ achieves a 3.37% detection rate improvement compared to Naive$_{200}$'s 3.99%. The Improved$_{200}$ variant achieves smaller improvements here because constraint-aware input generation produces less diverse arithmetic inputs, concentrating on boundary values rather than exploring the full input range where arithmetic mutations might create more varied outputs. Given that `Math` mutations comprise 59.1% of all mutants, this difference significantly impacts overall mutation scores. To counteract these effects, Improved variants could use higher `tries` settings to maintain the same non-boundary coverage as Naive. Alternatively, more sophisticated input selection strategies could be used to better balance boundary and non-boundary testing even at lower `tries` settings, thus avoiding the runtime cost of higher `tries`.

Results for the 10 and 50 `tries` variants of Naive and Improved generally follow the same trends as those for the listed variants with 200 `tries`, albeit with smaller detection rate improvements over Initial. The exception is `Math` mutation detection, where Improved$_{10}$ achieves only 1.1% improvement compared to 2.8% for Naive$_{10}$. Since `Math` mutations comprise 59.1% of all mutants, this explains the low overall detection rate of Improved$_{10}$ on eqbench-es-∗ projects observed in rows 1–3 of Figure 4. With 50 `tries`, the `Math` detection gap is noticeably smaller (Improved$_{50}$ achieves 3.1% versus Naive$_{50}$'s 3.6%), confirming that limited `tries` constrain arithmetic diversity only when boundary testing consumes most attempts. Full results for all variants are available in our replication package [32].

Answer to RQ1: Test generalization improves mutation detection across all evaluated projects, achieving absolute improvements of 1–4 percentage points depending on project type and generalization strategy. eqbench-es-∗ projects show the largest improvements (1.2–3.9pp, or 2.4–8.2% relative increase). commons-utils-es-∗ improves by 0.82–1.33pp (1.4–2.3% relative increase) and commons-utils-dev shows minimal improvement (0.05–0.07pp) due to its high baseline detection rate of 80.4%. Effectiveness varies by mutation operator: constraint-aware generation (Improved) excels at detecting boundary-related mutations like `ConditionalsBoundary` (+2.55pp), while random generation (Naive) performs better on `Math` mutations (+3.99pp vs +3.37pp) due to greater arithmetic diversity.

4.3 RQ2: How does constraint complexity affect random versus constraint-aware input generation?

As discussed in Section 4.2.1, Naive outperforms Improved on eqbench-es-∗ projects. However, commons-utils-es-∗ projects show Improved outperforming Naive in 7 of 9 cases. To better understand these contrasting results, RQ2 examines how constraint complexity differs across projects and how it affects Naive versus Improved. As described in Section 3.4.3, Improved tests encode simple in-/equalities on numeric and boolean variables or constants during input value generation. More complex constraints are not encoded during Improved input generation — and no constraints are encoded by Naive — but are still enforced during input filtering which takes place after input generation.

Table 4 shows the model properties of mutants that are (not) detected by Teralizer's Improved$_{200}$ generalization variant. Models represent the constraints that inputs must satisfy to reach each mutant along a specific execution path. We measure model complexity through operation count (total operators) and constraint count (individual boolean conditions), while tracking which percentage of constraints Improved can encode during input generation versus enforce through post-generation filtering. For instance, consider `(((a < 0) && (a == (b + 1))) && c)`. This model contains three constraints: `a < 0`, `a == (b + 1)`, and `c`. Improved encodes the simple comparison `a < 0` and the boolean variable `c` in the created input value generation code, but encodes `a == (b + 1)` only in the input value filtering code because it contains the compound term `b + 1`. Thus, Improved uses 2 of 3 total constraints for input value generation (66.7% utilization), and the model contains 5 operators: `<`, `&&`, `==`, `+`, and another `&&`.

Undetected mutants have more complex models than detected ones across all evaluated projects. Operation counts for undetected mutants are 1.2–3× higher: eqbench-es-∗ projects show mean counts of 218–231 operations for undetected mutants versus 138–147 operations for detected mutants, while commons-utils-es-∗ show even larger gaps with 389–507 versus 290–468 operations. Constraint counts follow similar patterns, with undetected mutants having 1.0–2.5× more constraints. Even though both Naive and Improved achieve better generalization outcomes for simpler constraints, more complex constraints have a stronger detrimental effect on Naive, which produces 2-2.5× as many `TooManyFilterMissesExceptions` as Improved, as discussed in more detail in Section 4.6.

Table 4. Model properties of mutants that are (not) detected by the IMPROVED$_{200}$ variant.

			Operations		Constraints		Constraints Used	
Project	Detected	Mutants	Mean	Median	Mean	Median	Mean	Median
eqbench-es-1s	yes	11,145	147	9	6	2	47%	80%
eqbench-es-1s	no	10,347	224	16	11	5	23%	50%
eqbench-es-10s	yes	11,658	139	9	6	2	62%	100%
eqbench-es-10s	no	9,999	231	15	8	2	57%	100%
eqbench-es-60s	yes	12,052	137	9	5	2	69%	100%
eqbench-es-60s	no	9,958	218	11	6	2	67%	100%
commons-utils-es-1s	yes	4,390	290	15	7	5	43%	84%
commons-utils-es-1s	no	3,183	389	45	12	6	11%	50%
commons-utils-es-10s	yes	4,660	467	23	6	5	46%	85%
commons-utils-es-10s	no	3,309	507	46	8	6	10%	56%
commons-utils-es-60s	yes	4,821	374	20	6	5	47%	85%
commons-utils-es-60s	no	3,288	423	41	10	6	11%	54%
commons-utils-dev	yes	4,193	107	11	4	4	25%	75%
commons-utils-dev	no	1,022	173	10	4	4	19%	75%

Constraint utilization rates show large differences across project types. eqbench-es-∗ achieve 47–70% mean constraint utilization for detected mutants, while commons-utils-es-∗ achieve only 25–47% mean utilization. The higher utilization in eqbench-es-∗ reflects their simpler constraint structures: these projects primarily use basic numeric comparisons that match IMPROVED's encoding capabilities. commons-utils-es-∗ projects contain more compound terms and mathematical functions that are not modeled by TERALIZER, reducing the percentage of constraints that can guide input generation.

These utilization differences explain the contrasting detection results. In eqbench-es-∗, simple constraints enable effective boundary targeting for IMPROVED, yet these same simple constraints make NAIVE's random generation viable. The higher constraint utilization even has detrimental effects on IMPROVED detection rates because the focus on boundary testing detracts from testing of intermediate values. As a result, detection rates for the very common `Math` mutations decrease, causing overall detection rates to go down despite detection rates for most other mutants increasing.

commons-utils-es-∗ projects present a different scenario. Complex constraints reduce IMPROVED's constraint utilization to 25–47%, causing generalized tests to generate inputs from broader ranges that overapproximate the true partition boundaries. As a result, fewer partition boundaries are accurately identified, and the number of generated inputs that need to be excluded during filtering increases. Nevertheless, constraint utilization still reduces `TooManyFilterMissesException` failures relative to NAIVE (Section 4.6), which enables IMPROVED variants to achieve higher mutation detection rates than NAIVE in 7 of 9 cases despite its `Math` mutation detection disadvantage.

Three paths emerge to further enhance IMPROVED's effectiveness. First, the `Math` mutation trade-off can be addressed through higher `tries` settings or balanced generation strategies that maintain boundary detection advantages while improving arithmetic coverage. Second, extending TERALIZER's constraint encoding support to handle more complex constraints would further increase utilization rates, thus enabling more effective constraint-aware input generation. However, encoding of non-boundary constraints would require custom input generators that are more capable than those provided by jqwik. Third, adaptive strategies could select generation approaches based on measured constraint complexity and mutation distribution, applying constraint-aware generation where it provides the largest benefit.

Answer to RQ2: Both input generation strategies perform better on simpler constraints, but Naive's effectiveness degrades more strongly as constraint complexity increases. On eqbench-es-∗ projects with simpler constraints, Naive outperforms Improved because random generation satisfies many constraints by chance, while Improved's boundary focus limits arithmetic diversity within the available `tries`, thus reducing `Math` mutation detection rates. On commons-utils-es-∗ projects with more complex constraints, Naive generates substantially more inputs that violate constraints, causing more `TooManyFilterMissesException` failures. Improved's constraint-aware generation reduces these failures, enabling it to outperform Naive in 7 of 9 cases despite its `Math` mutation disadvantage. Balancing boundary and non-boundary testing could combine the advantages of both strategies.

4.4 RQ3: To which degree does generalization affect the size and runtime of the target test suites?

Teralizer transforms conventional JUnit tests into property-based jqwik tests. While this transformation improves mutation detection (Section 4.2), it also affects test suite characteristics in several ways:

(1) the number of tests in the test suite (Section 4.4.1),
(2) the number of lines of code in the test suite (Section 4.4.2),
(3) the execution time of the test suite (Section 4.4.3).

Section 4.4.1 reveals how test architecture determines whether added generalized tests can be compensated by original test removals. Section 4.4.2 documents how Teralizer's constraint encoding and test isolation increase lines of code in the test suite. Section 4.4.3 analyzes runtime patterns, showing that costs stem primarily from property-based testing overhead and `tries` repetition. We focus on the results of Naive$_{200}$ and Improved$_{200}$ as they best represent the current capabilities of Teralizer. The results for variants with fewer `tries` follow similar trends, albeit with overall smaller effects on the measured metrics. Full results for all variants are available in our replication package [32].

4.4.1 Number of Tests in the Test Suite. As described in Section 3.5, Teralizer aims to remove any original and generalized tests that do not contribute unique fault detection capability during its test suite reduction stage. In an ideal scenario, all added property-based tests are compensated by removed original tests. However, the effectiveness of test suite reduction depends strongly on overall test architecture and mutation detection capabilities of the Original test suite. Table 5 quantifies the observed changes for the Naive$_{200}$ and Improved$_{200}$ generalization variants. The number of added tests is between 174–211 (3.5–4.4% of Original test suite size) for the eqbench-es-∗ projects, between 60–75 (2.2–2.8%) for the commons-utils-es-∗ projects, and 3 (0.4%) for the commons-utils-dev project.

Improved$_{200}$ generally adds a larger number of tests than Naive$_{200}$. This is because more of the tests that are created by Naive$_{200}$ are excluded due to `TooManyFilterMissesExceptions` and, therefore, not retained in the final test suite (as evaluated in Section 4.6). Furthermore, the number of added tests is much smaller than the total number of generalized tests that are created and evaluated by Teralizer for both Naive$_{200}$ as well as Improved$_{200}$ because only tests that measurably increase the mutation score of the test suite are retained. In total, Teralizer generates 21,478 candidate generalizations across the listed project and generalization variants. However, test suite reduction retains only 1,555 (7.2%) generalized tests that demonstrably improve mutation detection results.

Even though Teralizer adds hundreds of tests to the generalized test suites, net test count changes remain minimal. Added tests are largely compensated by removed tests in the eqbench-es-∗ and commons-utils-es-∗ projects which use EvoSuite-generated test suites. As a result, total test suite size only increases by 0–1 test cases (0–0.04% of Original test suite size) for these projects. The commons-utils-dev project sees less compensation success: none of the tests for

Table 5. Number of tests before and after generalization, with changes, per project.

		Tests					
Project	Variant	Before	Added	Removed	After	Delta	Delta %
eqbench-es-1s	NAIVE$_{200}$	4,718	177	177	4,718	+0	+0.0%
eqbench-es-1s	IMPROVED$_{200}$	4,718	206	206	4,718	+0	+0.0%
eqbench-es-10s	NAIVE$_{200}$	4,875	174	173	4,876	+1	+0.0%
eqbench-es-10s	IMPROVED$_{200}$	4,875	211	210	4,876	+1	+0.0%
eqbench-es-60s	NAIVE$_{200}$	4,974	174	174	4,974	+0	+0.0%
eqbench-es-60s	IMPROVED$_{200}$	4,974	210	210	4,974	+0	+0.0%
commons-utils-es-1s	NAIVE$_{200}$	2,481	60	59	2,482	+1	+0.0%
commons-utils-es-1s	IMPROVED$_{200}$	2,481	69	68	2,482	+1	+0.0%
commons-utils-es-10s	NAIVE$_{200}$	2,738	63	62	2,739	+1	+0.0%
commons-utils-es-10s	IMPROVED$_{200}$	2,738	70	69	2,739	+1	+0.0%
commons-utils-es-60s	NAIVE$_{200}$	2,735	60	59	2,736	+1	+0.0%
commons-utils-es-60s	IMPROVED$_{200}$	2,735	75	74	2,736	+1	+0.0%
commons-utils-dev	NAIVE$_{200}$	725	3	0	728	+3	+0.4%
commons-utils-dev	IMPROVED$_{200}$	725	3	0	728	+3	+0.4%

which generalizations are added can be removed. This is because an ORIGINAL test can only be removed if generalized tests are created for all assertions in the test (Section 3.5). In projects with EVOSUITE-generated tests, this requirement is generally satisfied because most tests only contain a single assertion. However, this requirement is more difficult to satisfy for commons-utils-dev because the developer-written tests often contain multiple assertions.

4.4.2 Lines of Code in the Test Suite. While test counts remain relatively stable due to test suite reduction, lines of code (LOC) increase across all projects and generalization variants. Table 6 shows increases of 31.5–58.7% for eqbench-es-∗, 18.8–29.9% for commons-utils-es-∗, and 4.9–5.3% for commons-utils-dev. These increases correlate with the number of added tests rather than net test count changes. For example, NAIVE$_{200}$ adds 177 generalized tests to eqbench-es-1s and removes all 177 corresponding original tests. However, the added tests increase test suite size by 11,780 LOC while the removed tests reduce LOC by only 1,127. This asymmetry stems from two characteristics of TERALIZER's current implementation. First, TERALIZER encodes constraints explicitly in the source code of the created property-based tests (Section 3.4.2). As a result, the LOC impact scales with parameter count and constraint complexity. IMPROVED variants also show higher per-test LOC because of their more sophisticated input generation logic. For example, commons-utils-dev's three generalized tests require 36 additional LOC with IMPROVED$_{200}$ compared to NAIVE$_{200}$ (9,018 vs. 8,982 total LOC). Second, test isolation creates duplication. TERALIZER creates new test classes for each generalized method, copying imports, setup/teardown methods, helper functions, class fields, etc. from original tests (Section 3.4.1). This prioritizes safety over LOC efficiency, avoiding unintended interactions between generalized and original tests.

The duplication overhead differs significantly between EVOSUITE-generated and developer-written tests. eqbench-es-∗ and commons-utils-es-∗ projects show similar overhead (66–67 vs 62–63 LOC per added test, respectively), reflecting the uniformity of EVOSUITE-generated tests. In contrast, commons-utils-dev shows substantially higher overhead (140 LOC per added test). This is because developer-written tests contain more shared setup code as well as multi-assertion architectures that hinder compensation through test removals. The higher increases observed in eqbench-es-∗ projects (31–59%) compared to commons-utils-es-∗ projects (19–30%) stem primarily from the retained generalized

Table 6. Number of test lines before and after generalization, with changes, per project.

		Lines					
Project	Variant	Before	Added	Removed	After	Delta	Delta %
eqbench-es-1s	NAIVE$_{200}$	30,989	11,780	1,127	41,642	+10,653	+34.4%
eqbench-es-1s	IMPROVED$_{200}$	30,989	19,019	1,302	48,706	+17,717	+57.2%
eqbench-es-10s	NAIVE$_{200}$	32,503	11,520	1,061	42,962	+10,459	+32.2%
eqbench-es-10s	IMPROVED$_{200}$	32,503	20,353	1,284	51,572	+19,069	+58.7%
eqbench-es-60s	NAIVE$_{200}$	33,510	11,623	1,069	44,064	+10,554	+31.5%
eqbench-es-60s	IMPROVED$_{200}$	33,510	20,288	1,285	52,513	+19,003	+56.7%
commons-utils-es-1s	NAIVE$_{200}$	16,563	3,733	359	19,937	+3,374	+20.4%
commons-utils-es-1s	IMPROVED$_{200}$	16,563	5,261	413	21,411	+4,848	+29.3%
commons-utils-es-10s	NAIVE$_{200}$	18,124	3,942	379	21,687	+3,563	+19.7%
commons-utils-es-10s	IMPROVED$_{200}$	18,124	5,423	421	23,126	+5,002	+27.6%
commons-utils-es-60s	NAIVE$_{200}$	17,886	3,723	361	21,248	+3,362	+18.8%
commons-utils-es-60s	IMPROVED$_{200}$	17,886	5,801	452	23,235	+5,349	+29.9%
commons-utils-dev	NAIVE$_{200}$	8,561	421	0	8,982	+421	+4.9%
commons-utils-dev	IMPROVED$_{200}$	8,561	457	0	9,018	+457	+5.3%

tests representing a larger fraction of the original test suite in eqbench-es-∗ (3.5–4.4%) compared to commons-utils-es-∗ (2.2–2.8%). The smaller difference between NAIVE and IMPROVED in commons-utils-es-∗ compared to eqbench-es-∗ likely reflects IMPROVED's lower constraint utilization (28.5% for commons-utils-es-∗ vs. 54.7% for eqbench-es-∗).

Both overhead sources represent implementation choices rather than inherent limitations. The primary reason TERALIZER avoids in-place transformation of tests is to keep changes isolated, preventing adverse effects on developer-written code and mutation testing (Section 3.4.1). Similarly, constraint encoding logic could be extracted to a library that abstracts implementation details. Only minor changes would remain then in the generalized test code: modified test annotations, parameterized inputs, and generalized assertions. The overall impact on test suite LOC would, therefore, be reduced while preserving the mutation detection benefits. We leave these improvements for future work.

4.4.3 Execution Time of the Test Suite. As shown in Table 7, generalization with NAIVE$_{200}$ and IMPROVED$_{200}$ increases overall test suite runtimes for all eqbench-es-∗ and commons-utils-es-∗ projects. More specifically, test suite runtimes show increases of 574.5–1210.0% for the eqbench-es-∗ projects, 444.1–2651.5% for the commons-utils-es-∗ projects, and 9.4–57.1% for the commons-utils-dev project. Runtime increases are primarily affected by the following factors:

(1) the number of tests added by the generalization,
(2) the number of tests removed by the test suite reduction,
(3) the number of `tries` used during property-based testing,
(4) the used generalization approach, i.e., NAIVE vs. IMPROVED generalization,
(5) the complexity of input partition constraints.

Added and Removed Tests. Execution of conventional JUnit tests has a lower runtime cost than execution of corresponding jqwik tests. More specifically, property-based tests created with the BASELINE generalization strategy take, on average, 149.56 milliseconds (ms) longer to execute than the corresponding ORIGINAL tests (as shown in Figure 5) which have a mean execution time of only 3.6 ms. Therefore, overall test suite execution time increases, on average, by

Table 7. Test suite runtime before and after generalization, with changes, per project.

		Runtime (in seconds)					
Project	Variant	Before	Added	Removed	After	Delta	Delta %
eqbench-es-1s	NAIVE$_{200}$	17.44	100.94	0.74	117.65	+100.20	+574.5%
eqbench-es-1s	IMPROVED$_{200}$	17.44	101.62	0.68	118.38	+100.94	+578.7%
eqbench-es-10s	NAIVE$_{200}$	16.70	106.19	0.66	122.23	+105.53	+632.0%
eqbench-es-10s	IMPROVED$_{200}$	16.70	139.64	0.56	155.77	+139.08	+832.9%
eqbench-es-60s	NAIVE$_{200}$	18.21	221.07	0.76	238.52	+220.31	+1210.0%
eqbench-es-60s	IMPROVED$_{200}$	18.21	124.68	0.69	142.20	+123.99	+681.0%
commons-utils-es-1s	NAIVE$_{200}$	4.31	114.42	0.09	118.64	+114.33	+2651.5%
commons-utils-es-1s	IMPROVED$_{200}$	4.31	29.49	0.14	33.66	+29.35	+680.5%
commons-utils-es-10s	NAIVE$_{200}$	7.40	148.54	0.14	155.80	+148.40	+2005.2%
commons-utils-es-10s	IMPROVED$_{200}$	7.40	71.50	0.21	78.70	+71.30	+963.3%
commons-utils-es-60s	NAIVE$_{200}$	6.30	122.11	0.07	128.34	+122.04	+1936.2%
commons-utils-es-60s	IMPROVED$_{200}$	6.30	28.07	0.08	34.29	+27.99	+444.1%
commons-utils-dev	NAIVE$_{200}$	7.95	4.54	0.00	12.48	+4.54	+57.1%
commons-utils-dev	IMPROVED$_{200}$	7.95	0.74	0.00	8.69	+0.74	+9.4%

at least 149.56 ms per jqwik test that is added during generalization, even if no new test inputs are exercised and the added generalized tests are compensated by removed original tests. Since these runtime increases are inherent to the use of jqwik, they are orthogonal to our specific generalization approach. Any manual or automated transformation of JUnit tests to jqwik tests incurs the same runtime overhead, and this overhead can only be reduced if fewer jqwik tests are created (e.g., through test suite reduction, as discussed in Section 3.5) or if the performance of jqwik is improved.

Number of `tries`. Increasing the number of `tries` directly increases the number of test inputs that need to be generated and exercised during property-based test execution. For example, as shown in Figure 5, execution time of NAIVE tests is, on average, 286.13 milliseconds (ms) longer per test than for ORIGINAL tests when using 10 `tries` (a +91.3% increase compared to the BASELINE overhead of 149.56 ms), 348.56 ms (+133.0%) longer with 50 `tries`, and 1136.21 ms (+659.7%) longer with 200 `tries`. Similarly, execution time of IMPROVED tests is 189.17 ms (+26.5%) longer with 10 `tries`, 246.85 ms (+65.1%) longer with 50 `tries`, and 395.26 ms (+164.3%) longer with 200 `tries`. While overall runtime increases as `tries` increase, the runtime cost per `try` decreases as `tries` increase. More specifically, NAIVE variants show per-`try` increases of 28.61 ms / 6.97 ms / 5.68 ms at 10 / 50 / 200 `tries`. IMPROVED variants show per-`try` increases of 18.92 ms / 4.94 ms / 1.98 ms at 10 / 50 / 200 `tries`. However, to attribute these observed improvements in per-`try` efficiency to any specific causes would require a more thorough microbenchmarking setup that properly accounts for confounding factors such as JVM warmup, which is beyond the scope of this evaluation.

NAIVE vs. IMPROVED Generalization. Runtime increases are generally larger for NAIVE than for IMPROVED. For example, NAIVE$_{200}$ and IMPROVED$_{200}$ both generalize the same three tests of commons-utils-dev (see Table 5). Nevertheless, as shown in Table 7, NAIVE$_{200}$ increases test suite runtime by 4.54 seconds (+57.1% compared to ORIGINAL) whereas IMPROVED$_{200}$ only increases runtime by 0.74 seconds (+9.4%). As described in Section 3.4.2, NAIVE selects inputs by first randomly generating values that match the parameter types of the MUT, and then filtering any values that do not satisfy the input specification. Especially for cases with restrictive constraints (e.g., `a == b && b == c`), this causes a

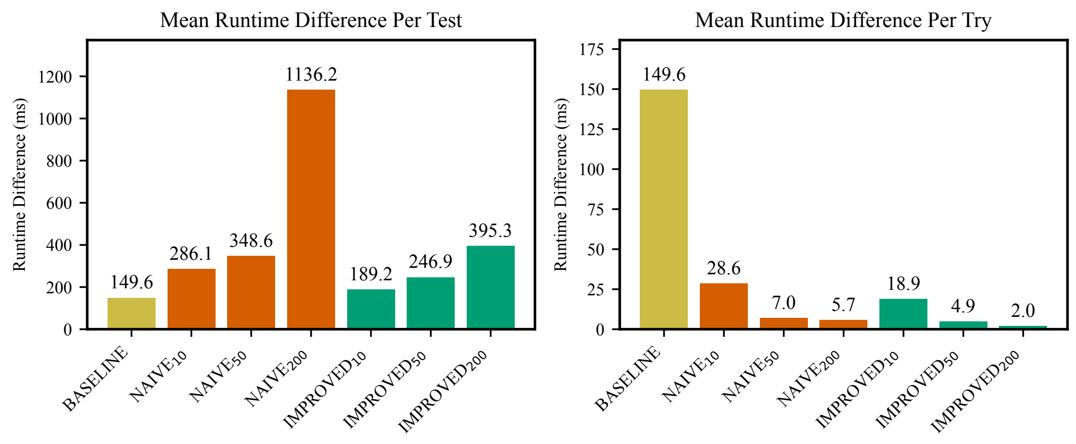


Fig. 5. Runtime comparison between original and generalized tests. The runtime differences measure how much longer generalized tests take to execute, on average, compared to corresponding original tests. We show the difference per test (left) and per try (right).

large runtime overhead because many filter-and-regenerate cycles are required until valid inputs are identified (or jqwik throws a `TooManyFilterMissesException`). Improved variants are less affected by this because they encode (some) input constraints during input generation (Section 3.4.2). As a result, fewer inputs need to be filtered and regenerated, which lessens the runtime impact of Improved generalization despite its more involved input generation process.

Complexity of Constraints. As complexity of input specifications increases, required runtime also increases. This is because more complex constraints cause more filter-and-regenerate cycles to occur during execution of property-based tests. While Naive is more strongly affected by this than Improved because it does not consider any constraints during value generation (as discussed in the previous paragraph), the issue also affects Improved generalization in cases where less than 100% of constraints can be used (for further details on constraint use, see Sections 3.4.2 and 4.3). For example, as shown in Table 5, Improved$_{200}$ adds 3 times as many tests (206 vs. 69) to eqbench-es-1s as it adds to commons-utils-es-1s, yet runtime increases are similar: +578.7% vs. +680.5%. This is because input specifications in eqbench-es-1s have fewer operations (mean: 159.1 vs. 208.0, median: 10 vs. 15) and constraints (mean: 6.5 vs. 7.5, median: 2 vs. 5) than in commons-utils-es-1s, and fewer of these constraints can be used during input value generation (mean: 42.9% vs. 61.5%, median: 80% vs. 100%). This suggests that improving support for complex input constraints would not only increase detection rates (as suggested in Section 4.3) but could also reduce the runtime cost of generalized tests.

Answer to RQ3: Test generalization consistently increases test suite LOC and runtime, whereas test count changes are highly dependent on test architecture. EvoSuite-generated test suites see near-complete compensation via test suite reduction (1,549 tests added, 1,541 removed). Developer-written tests resist compensation due to multi-assertion architectures (6 added, 0 removed). LOC increases by 4.9–58.7% due to explicit constraint encoding and test isolation. Improved shows larger increases than Naive because of its more sophisticated input generation logic. Runtime increases by 9.4–2,651.5% across projects and `tries` settings, reflecting property-based testing overhead, i.e., jqwik framework cost plus increased number of tested inputs. Naive shows larger runtime increases than Improved because random generation requires more filter-and-regenerate cycles to satisfy constraints.

4.5 RQ4: How efficient is test generalization compared to test generation?

We measured TERALIZER's runtime across the seven generalization strategies to assess viability compared to existing automated testing tools. Since no existing, publicly available tools perform automated test generalization, we compared efficiency against EVOSUITE [29]. SHARED processing stages (Stages 1–3) were executed only once per project, with specifications reused across generalization strategies. Processing took 3.4 hours for commons-utils-dev, 8.2 / 9.1 / 9.8 hours for the commons-utils-es-∗ variants, and 24.8 / 28.3 / 30.9 hours for the eqbench-es-∗ variants (Section 4.5.1). Pareto analysis (Section 4.5.2) shows that combining low search budget EVOSUITE generation with TERALIZER generalization can achieve better detection-to-runtime ratios than simply running EVOSUITE with higher search budgets.

4.5.1 Execution Time of TERALIZER. Our runtime evaluation results reveal that test suite reduction via mutation testing dominates the overall runtime cost of TERALIZER. More specifically, Figure 6 shows that Stage 5 (test suite reduction) consumes 1,110–35,538 seconds (59.1–95.7% of total processing time) across projects and generalization strategies. In contrast, Stages 1 + 2 (project analysis) require only 82–453 seconds (1.2–6.5%), Stage 3 (specification extraction) requires 60–2,776 seconds (1.5–36.4%), and Stage 4 (generalized test creation) requires 5–178 seconds (0.1–2.3%).

Stage 1–4 Runtimes. The first four processing stages complete efficiently despite containing the core generalization logic. Project analysis (Stage 1 + 2) executes the ORIGINAL test suite and performs simple analyses on the JUnit reports and test code, both of which create only minimal overhead beyond the test suite execution. Specification extraction (Stage 3) uses SPF to concretely execute tests using single-path symbolic analysis (Section 3.3). While this introduces some overhead because JPF/ SPF is less optimized than production-ready JVMs (JPF itself runs inside a host JVM [64]), the cost is comparatively small at, on average, ca. 5× the runtime of an ORIGINAL test suite execution (mean: 1020s vs. 221s). Generalized test creation cost (Stage 4) is even smaller at a one-time cost of 5–177 seconds per variant. This is because test creation only performs syntactic replacements, e.g., converting JUnit annotations to jqwik ones, wrapping input constraint encodings in `TestParameters` classes, and replacing expected values in assertions (Section 3.4).

Stage 5 Runtimes. Test suite reduction costs are substantially larger than the costs of the preceding stages due to the high runtime cost of mutation testing. Mutation testing of the ORIGINAL test suite takes, on average, 8× as long as ORIGINAL test suite execution without mutation testing (mean: 1,833s vs. 221s). Mutation testing of the generalized test suites has even higher runtime requirements (mean: 4,351s), which is for largely the same reasons that jqwik tests take longer to execute than JUnit tests: jqwik overhead, larger number of tested inputs, as well as filter-and-regenerate cycles which occur more often for NAIVE variants and in the presence of more complex input constraints (Section 4.4.3). Thus, mutation testing consumes more than 99% of Stage 5 runtimes across projects and generalization variants, and an average of 82.9% of the full generalization pipeline. The small remainder of Stage 5 runtimes falls to the collection of coverage reports as well as processing of coverage and mutation reports to exclude non-contributing tests.

While test suite reduction could be skipped to reduce the one-time cost of generalization, this would significantly increase the execution times of the generalized test suites. After all, the primary purpose of test suite reduction is to remove generalized tests that do not improve fault detection, thus avoiding the high runtime cost associated with property-based test execution (Section 4.4.3). In our evaluation, this filtering reduces the number of retained generalized tests from 21,478 to 1,555 across all projects and test suite variants (Section 4.4.1), thus demonstrating the impact that filtering has on the size and runtime of the generalized test suites. Even though there is a non-negligible one-time cost associated with this, that cost amortizes over time compared to a longer-running test suite that incurs further costs

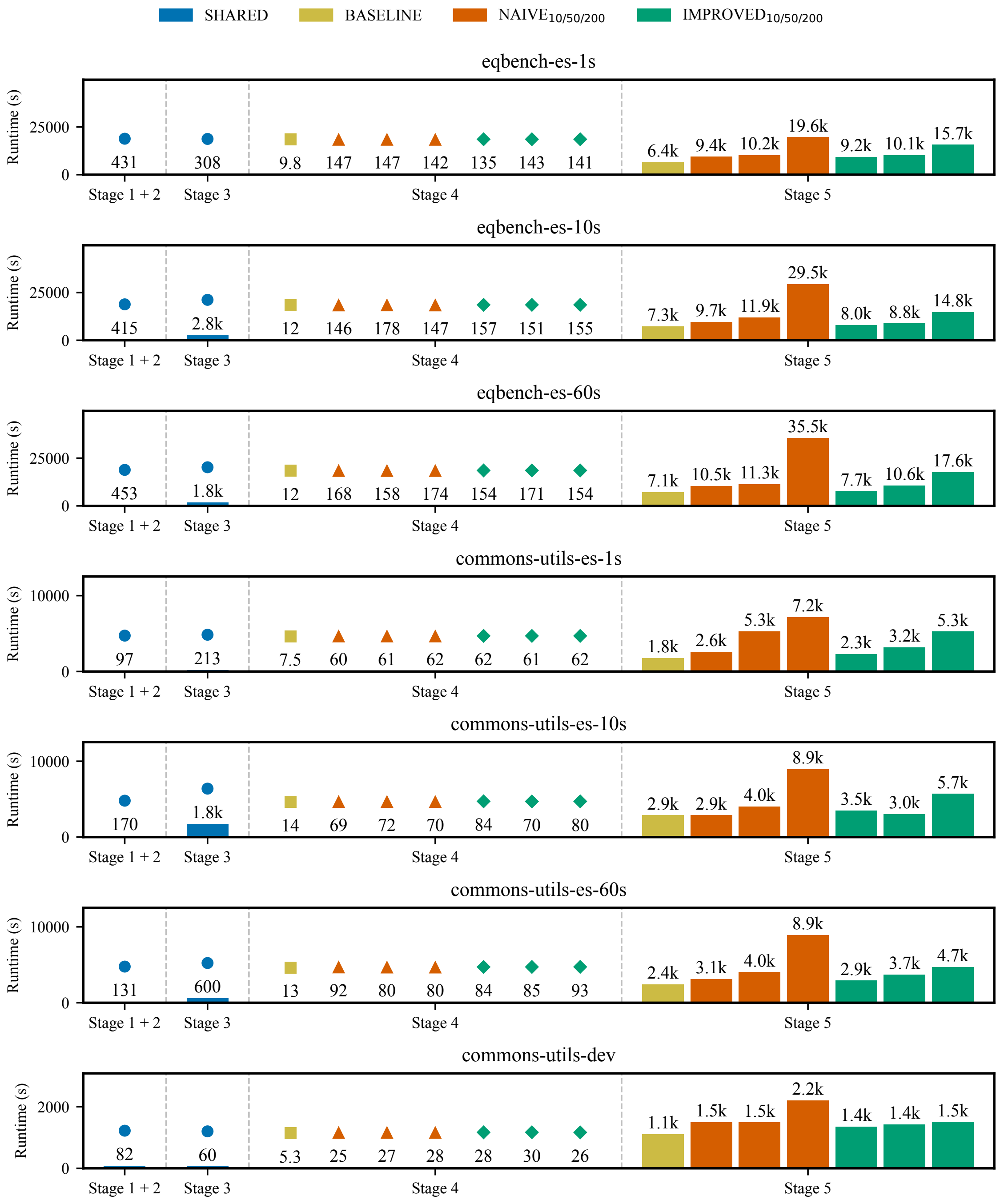


Fig. 6. Teralizer runtimes per project, processing stage, and generalization strategy. Stages 1–3 are SHARED stages that are only executed once. Stages 4 and 5 are executed once per generalization strategy, i.e., BASELINE, NAIVE$_{10/50/200}$, and IMPROVED$_{10/50/200}$.

with every test suite execution. Future optimization efforts could target filter efficiency through faster mutation testing approaches or lightweight pre-filtering heuristics that identify likely-beneficial candidates without full mutation testing.

4.5.2 Efficiency of Teralizer vs. EvoSuite. Our evaluation results show that combining EvoSuite's test generation with Teralizer's test generalization can achieve better detection-to-runtime ratios than simply increasing EvoSuite search budgets. Figure 7 identifies which tool configurations provide the best detection-to-runtime trade-offs through Pareto analysis. Configurations on the frontier represent optimal choices: for any given runtime budget, they achieve the highest detection rate. Configurations outside the frontier are dominated by the Pareto-optimal points.

For the eqbench-es-∗ projects, 2 of 3 EvoSuite search budget settings are Pareto optimal (Pareto points #1 and #2 in Table 8) due to their comparatively low runtime cost. EvoSuite generation with a 60 seconds per-class search budget falls outside the Pareto frontier with a detection rate of 51.6% and a runtime cost of 55,075 seconds. In the runtime dimension, it is dominated by 1-second EvoSuite generation combined with Naive_{50} generalization (Pareto point #5 in Table 8), which achieves 51.7% detection in 37,532 seconds, i.e., 0.1 percentage points higher detection with 31.9% lower runtime. In the mutation detection dimension, it is dominated by 10-second EvoSuite generation with Improved_{200} generalization (Pareto point #9), which reaches a 53.8% detection rate in 48,269 seconds, i.e., a detection improvement of 2.2 percentage points achieved in 12.4% less runtime. Increasing the runtime beyond this comparison point achieves higher detection rates in the generalized test suites that further extend the Pareto frontier (Pareto points #10–#14).

Efficiency improvements of EvoSuite + Teralizer combinations over EvoSuite search budget increases are less pronounced for the commons-utils-es-∗ projects. Here, all three EvoSuite search budget settings are Pareto optimal (Pareto points #1, #2, and #4 in Table 9). Nevertheless, generalization via Teralizer contributes 8 additional points to the Pareto frontier. Specifically, the combination of 1-second EvoSuite generation and Improved_{10} generalization (Pareto point #3) produces a Pareto optimal result that has higher detection rate and runtime cost than #2 but lower detection rate and runtime cost than #4. The 7 remaining Pareto points #5–#11 again extend the Pareto frontier toward higher detection rates at higher runtime cost, reflecting the increased detection rates achievable via generalization before reaching a plateau at around 1.0–1.3 percentage points above the corresponding EvoSuite results (Section 4.2.)

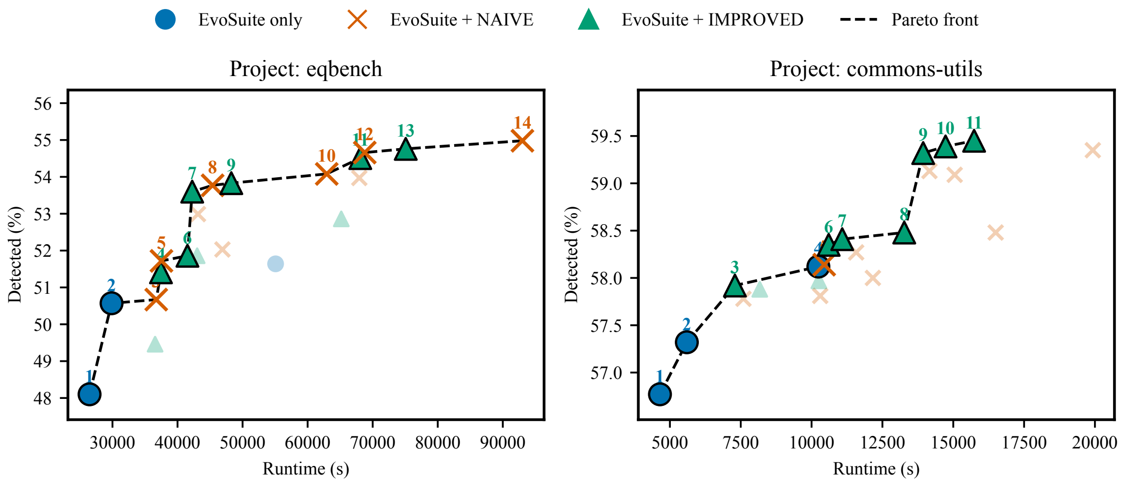


Fig. 7. Pareto fronts for EvoSuite and Teralizer variants across projects.

Table 8. Pareto points for project: eqbench.

Pt.	EvoSuite	Teralizer	Det. %	Runtime (s)
1	1s	-	48.1	26,479
2	10s	-	50.6	29,861
3	1s	NAIVE$_{10}$	50.7	36,728
4	1s	IMPROVED$_{50}$	51.4	37,457
5	1s	NAIVE$_{50}$	51.7	37,532
6	10s	IMPROVED$_{10}$	51.9	41,525
7	10s	IMPROVED$_{50}$	53.6	42,256
8	10s	NAIVE$_{50}$	53.8	45,398
9	10s	IMPROVED$_{200}$	53.8	48,269
10	10s	NAIVE$_{200}$	54.1	62,938
11	60s	IMPROVED$_{50}$	54.5	68,093
12	60s	NAIVE$_{50}$	54.7	68,782
13	60s	IMPROVED$_{200}$	54.8	75,081
14	60s	NAIVE$_{200}$	55.0	93,017

Table 9. Pareto points for project: commons-utils.

Pt.	EvoSuite	Teralizer	Det. %	Runtime (s)
1	1s	-	56.8	4,649
2	10s	-	57.3	5,597
3	1s	IMPROVED$_{10}$	57.9	7,294
4	60s	-	58.1	10,240
5	10s	NAIVE$_{10}$	58.1	10,445
6	10s	IMPROVED$_{50}$	58.4	10,603
7	10s	IMPROVED$_{10}$	58.4	11,082
8	10s	IMPROVED$_{200}$	58.5	13,270
9	60s	IMPROVED$_{10}$	59.3	13,939
10	60s	IMPROVED$_{50}$	59.4	14,728
11	60s	IMPROVED$_{200}$	59.5	15,736

Naive and Improved both produce 6 Pareto points from their 9 evaluated configurations for the eqbench-es-$*$ project (3 EvoSuite search budgets, each combined with 3 `tries` settings). For commons-utils-es-$*$, Improved has 7 of 9 of results on the Pareto frontier, compared to 1 of 9 results of Naive. The underlying causes were previously discussed in RQ2 and RQ3 (Section 4.3 and Section 4.4): less complex constraints in eqbench-es-$*$ favor Naive, enabling it to be competitive with Improved despite its simpler input generation approach. In contrast, constraints are more complex for the commons-utils-es-$*$ projects. This increases the mutation detection rate and runtime advantage that Improved has over Naive because the more sophisticated input value generation avoids `TooManyFilterMissesExceptions`.

Answer to RQ4: Test generalization requires a substantial one-time investment (3.4–30.9 hours per project) which is primarily due to Stage 5 mutation testing for test suite reduction (59.1–95.7% of total time). Reduction is necessary to filter 21,478 candidate generalizations down to 1,555 retained tests, thus avoiding ongoing test suite execution overhead. Despite the high processing costs, Pareto analysis shows that combining short EvoSuite generation with Teralizer generalization achieves better detection-to-runtime ratios than longer EvoSuite generation alone for multiple evaluated configurations. This stems from complementary optimization targets: EvoSuite optimizes for breadth (coverage), while Teralizer optimizes for depth (thorough testing of discovered paths).

4.6 RQ5: What are the causes of unsuccessful generalization attempts under controlled conditions?

While RQ1–4 demonstrate that Teralizer can improve mutation detection rates and operate within practical time constraints, our evaluation also revealed that many generalization attempts do not succeed. In RQ5, we first present the results for the primary evaluation dataset, i.e., for the eqbench-es-$*$ and commons-utils-$*$ projects, to establish a baseline under controlled conditions that align with current tool capabilities (as explained in Section 4.1). In RQ6, we then investigate the results for the RepoReapers projects to better understand real-world applicability challenges.

As explained in Section 3, there are two different types of causes based on which Teralizer may exclude individual tests, assertions, or generalizations from further processing: filtering and failures. Filtering preemptively excludes cases that are beyond the current capabilities of Teralizer to focus on suitable generalization candidates. Even though a single filter rejection is enough to exclude a given test, assertion, or generalization, Teralizer generally collects responses from all applicable filters to enable a more robust analysis of filtering causes. Despite preemptive filtering, some processing attempts fail due to exceptions that are thrown at runtime, prompting Teralizer to exclude the corresponding test, assertion or generalization from further processing once such an exception occurs.

Table 10 quantifies inclusion and exclusion rates of tests, assertions, and generalizations while distinguishing between filtering-based and failure-based exclusions. A more fine-grained overview of filtering results is shown in Table 11. Filters that defer do not cast a vote because they have insufficient information to make an accept or reject decision.

Test-level Exclusions. Overall, 19,306 of 23,246 Original tests (83.1%) across all variants of the primary evaluation dataset remain included. Filtering excludes 3,933 tests (16.9%) due to filter rejections. The largest number of tests is rejected by the `NoAssertions` filter (10.3% of tests rejected), followed by the `NonPassingTest` filter (6.6%) and the `TestType` filter (0.8%). Null pointer dereferences that occur during project analysis exclude 7 additional tests (0.0%).

Of the 1,527 `NonPassingTest` rejections, 132 are tests that fail after disabling EvoSuite's isolation and reproducibility features. These features had to be disabled due to incompatibilities which caused PIT crashes during mutation testing. However, removal of these features causes EvoSuite-generated tests that rely on system time or specific environmental conditions to fail. Because PIT only supports class-level exclusion, 1,395 additional passing tests in the same classes are also excluded as a side effect, amplifying the impact of individual test failures by over 10×.

All 180 `TestType` rejections are caused by the presence of @ParameterizedTest annotations in the commons-utils-dev project. As described in Section 3.1, Teralizer currently supports only standard @Test annotations, forcing it to exclude @ParameterizedTest, @RepeatedTest, and other specialized test types from processing.

The `NoAssertions` filter operates on 21,532 tests that remain after exclusions due to test-level failures (7 tests) and rejections by the preceding filters (1527 + 180 tests). From this subset, 2,226 tests are rejected because they do not directly contain any assertions in the test method. In EvoSuite-generated test suites, all `NoAssertions` exclusions are genuinely assertion-free tests that pass if no exception occurs during test execution. In contrast, 69 of 80 `NoAssertions` exclusions (86.3%) in the developer-written commons-utils-dev test suite are false positives: these tests contain assertions in helper methods called from the test method. However, Teralizer's current static analysis only examines the top-level test method, which causes it to miss these delegated assertion calls (as described in Section 3.1).

Assertion-level Exclusions. Across the 28,923 identified assertions within the primary evaluation dataset, a total of 13,836 (47.8%) are included and 15,087 (52.2%) are excluded. Of these exclusions, 12,092 are the result of filtering rejections whereas 2,995 stem from failures during specification extraction via SPF.

Causes for the 2,995 assertion-level failures include SPF errors, Teralizer errors, and exceeded analysis limits. SPF exceptions constitute 51.4% of failures (1,540 assertions). They occur primarily due to missing models for native methods in the current implementation of SPF and, to a lesser extent, due to implementation bugs in SPF/JPF. Exceeded analysis limits account for 45.3% of failures (1,358 assertions): 790 (26.4%) SPF runs are interrupted after exceeding the maximum specification size, and 524 (17.5%) runs exceed the maximum depth limit (Listing 4). Both of these aim to avoid timeouts and memory exhaustion in the presence of complex control flows or complex constraints. Further failures are due to timeouts (0.9%, 28 assertions) and out-of-memory errors (0.5%, 16 assertions) which evade preemptive detection via the two preceding measures. NullPointerExceptions represent the remaining 3.2% of failures (97 assertions).

A total of 5.5% of identified assertions (1,597) are rejected by the `ExcludedTest` filter because they belong to tests that are already excluded at the test level. This cascading effect ensures consistency across processing stages.

Table 10. Exclusion results for tests, assertions, and generalizations in the commons-utils-∗ and eqbench-es-∗ projects.

				Excluded	
Strategy	Level	Total	Included	Filtering	Failures
All	Test	23,246	19,306 (83.1%)	3,933 (16.9%)	7 (0.0%)
All	Assertion	28,923	13,836 (47.8%)	12,092 (41.8%)	2,995 (10.4%)
Baseline	Generalization	13,836	13,814 (99.8%)	22 (0.2%)	0 (0.0%)
Naive_{10}	Generalization	13,836	10,743 (77.6%)	3,061 (22.1%)	32 (0.2%)
Naive_{50}	Generalization	13,836	9,964 (72.0%)	3,840 (27.8%)	32 (0.2%)
Naive_{200}	Generalization	13,836	9,881 (71.4%)	3,923 (28.4%)	32 (0.2%)
Improved_{10}	Generalization	13,836	11,788 (85.2%)	2,016 (14.6%)	32 (0.2%)
Improved_{50}	Generalization	13,836	11,660 (84.3%)	2,144 (15.5%)	32 (0.2%)
Improved_{200}	Generalization	13,836	11,597 (83.8%)	2,207 (16.0%)	32 (0.2%)

Table 11. Filtering results for tests and assertions in the commons-utils-∗ and eqbench-es-∗ projects.

Level	Filter Name	Total	Accept	Defer	Reject
Test	NonPassingTest	23,246	21,719 (93.4%)	-	1,527 (6.6%)
Test	TestType	23,246	23,066 (99.2%)	-	180 (0.8%)
Test	NoAssertions	21,532	19,306 (89.7%)	-	2,226 (10.3%)
Assertion	AssertionType	28,923	28,180 (97.4%)	-	743 (2.6%)
Assertion	ExcludedTest	28,923	27,326 (94.5%)	-	1,597 (5.5%)
Assertion	MissingValue	28,923	21,766 (75.3%)	-	7,157 (24.7%)
Assertion	ParameterType	28,923	17,835 (61.7%)	6,630 (22.9%)	4,458 (15.4%)
Assertion	VoidReturnType	28,923	21,763 (75.2%)	7,157 (24.7%)	3 (0.0%)

Another 2.6% of assertions (743) are rejected by the `AssertionType` filter. Excluded assertions comprise reference equality checks (assertSame: 142, assertNotSame: 79), null checks (assertNull: 124, assertNotNull: 86), array comparisons (assertArrayEquals: 207), inequality assertions (assertNotEquals: 54), type checks (assertInstanceOf: 18), and explicit failures (fail: 33). These assertions largely involve data types that are not supported by current symbolic analysis.

The `MissingValue` filter excludes 24.7% of assertions. A rejection occurs when Teralizer's static analysis cannot identify which method call represents the method under test (MUT) or when the declaration of the MUT cannot be resolved by Spoon (Section 3.2). Missing values also cause `ParameterType` and `VoidReturnType` filters to defer.

The `ParameterType` filter rejects 15.4% of assertions where none of the tested method's parameters have generalizable types (numeric or boolean). Deferral numbers are lower than `MissingValue` rejections because, in some cases, Teralizer can infer parameter types from the call site of the MUT even if the full method declaration cannot be resolved.

Finally, the `VoidReturnType` filter rejects 3 assertions for tested methods with void return types. The current implementation of Teralizer does not support such methods because no output specification can be inferred for them.

Generalization-level Exclusions. All filtering-based generalization exclusions in Table 10 are due to `NonPassingTest` rejections. Baseline generalizations are affected by such rejections in 0.2% of cases (22 of 13,836 generalized tests). This low rejection rate indicates that the transformation to jqwik tests is largely successful. The 22 test failures that force `NonPassingTest` to reject occur for tests in the commons-utils-dev project that call MUTs or assertions within a loop.

This is problematic because TERALIZER's current implementation does not properly account for loops. As a result, the BASELINE generalization strategy replaces the `expected` value of assertions within loops with the concrete input value of the first loop iteration, which commonly causes later loop iterations to fail with an `AssertionError`.

NAIVE shows 3,061–3,923 (22.1–28.4%) filtering-based exclusions. The primary cause is `TooManyFilterMisses-Exceptions`, which occur 2,233 times (16.1%) for NAIVE_{10}, 2,938 (21.2%) for NAIVE_{50}, and 3,005 (21.7%) for NAIVE_{200}. Prevalence increases with higher `tries` as NAIVE struggles to produce enough valid inputs, especially for more complex constraints (Section 4.3). The remaining filtering-based exclusions are due to inaccurate input/output specifications which cause exceptions during assertion checking (`AssertionFailedError`, 729 / 803 / 819 exclusions by $\text{NAIVE}_{10/50/200}$) or general test execution (`ArithmeticException`, 99 / 99 / 99 exclusions). Underlying causes include (i) assertions in loops, (ii) assertions in transitively called methods, (iii) MUT calls within loops, and (iv) implicit preconditions. While (i)–(iii) are limitations of TERALIZER, (iv) can indicate unintended behavior such as potential divisions by 0 or under-/overflows that evade detection by the ORIGINAL tests but are identified by the generalized tests.

IMPROVED generalization strategies reduce overall exclusion rates from filtering-based rejections to 2,016–2,207 (14.6–16.0%) through constraint-aware input value generation (Section 4.3). Specifically, `TooManyFilterMissesExceptions` decrease from 2,223 to 1,189 (−46.5%) for NAIVE_{10} vs. IMPROVED_{10}, from 2,938 to 1,223 (−58.4%) for NAIVE_{50} vs. IMPROVED_{50}, and from 3,005 to 1,265 (−57.9%) for NAIVE_{200} vs. IMPROVED_{200}. This clearly demonstrates that IMPROVED input generation is more effective at producing inputs that satisfy identified input constraints. Because more generalizations pass early input filtering, the number of test failures due to later `AssertionFailedErrors` and `ArithmeticExceptions` often increases, albeit to a lesser degree. For NAIVE_{10} vs. IMPROVED_{10}, the number changes from 828 to 827 (−0.1%), for NAIVE_{50} vs. IMPROVED_{50} from 902 to 921 (+2.1%), and for NAIVE_{200} vs. IMPROVED_{200} from 918 to 942 (+2.6%).

Project-specific patterns show how test characteristics and constraint complexity affect generalization success. Developer-written tests in commons-utils-dev exhibit 11.5–14.2% `TooManyFilterMissesExceptions` and 24.7–25.7% from inaccurate specifications. EVOSUITE-generated tests in commons-utils-es-∗ show 14.9–15.7% and 12.7–15.3%, respectively, while EVOSUITE-generated tests in eqbench-es-∗ show 5.7–6.0% and 1.3%. The 2× higher inaccurate specification rate in commons-utils-dev compared to commons-utils-es-∗ reflects test construction differences: EVOSUITE-generated tests avoid loops and do not invoke assertions through helper methods, while the developer-written tests commonly use both patterns. The higher rates in commons-utils-es-∗ tests compared to eqbench-es-∗ tests — despite identical test construction patterns — reflects the impact of constraint complexity (Section 4.3).

Beyond filter rejections, NAIVE and IMPROVED fail 32 generalizations to avoid "code too large" errors. This error occurs when a method's bytecode exceeds 64KB, Java's hard limit for method size. Generalized tests can exceed this limit when specifications contain many complex constraints, necessitating preemptive exclusion of the largest specifications.

Answer to RQ5: Of 28,923 identified assertions, 9,881–13,836 (34.2–47.8%) are successfully generalized across the three strategies. Most exclusions occur at the assertion level, primarily due to static analysis limitations in identifying tested methods (24.7%) and the presence of parameter types that cannot be accurately modeled by current symbolic analysis (15.4%). SPF errors and exceeded analysis limits exclude 5.3% and 4.7% of assertions, respectively. Test-level filtering excludes 5.5% of assertions due to non-passing ORIGINAL tests. NAIVE generalized tests pass in 71.4–77.6% of cases, with failures primarily due to `TooManyFilterMissesExceptions` (16.1–21.7%) and inaccurate specifications in the presence of loops and interprocedural control flow in tests (6.0–6.6%). IMPROVED increases pass rates to 83.8–85.2% by reducing filter misses by 46.5–58.4% through constraint-aware input generation.

4.7 RQ6: What are the causes of unsuccessful generalization attempts under real-world conditions?

RQ5 established exclusion causes under controlled conditions. RQ6 now examines how Teralizer performs on 632 Java projects from the RepoReapers dataset (Section 4.1) to identify generalization barriers in real-world projects. Section 4.7.1 covers project-level exclusions. Section 4.7.2 examines test, assertion, and generalization exclusions, comparing exclusion rates to controlled settings. Full project-level exclusions and partial test, assertion, and generalization exclusions are interconnected: when filtering or failures exclude all tests, assertions, or generalizations in a project, the project is excluded. Understanding exclusion patterns guides future work toward addressing the most impactful limitations.

4.7.1 Project-Level Exclusions. Only 11 of 632 projects (1.7%) successfully complete all five processing stages (Table 12), revealing substantial barriers to real-world applicability. To understand where and why processing fails, we examine exclusions stage by stage, distinguishing between internal causes (caused by configured resource limits or limitations of Teralizer), external causes (caused by Teralizer's dependencies: JUnit, Spoon, JPF/SPF, JaCoCo, and PIT), and mixed causes (influenced by both internal and external factors). This reveals which barriers are addressable through future improvements of Teralizer and which ones reflect less actionable limitations in Teralizer's dependencies.

Table 12. Project-level exclusions by stage and cause for the Improved$_{200}$ generalization strategy in RepoReapers projects. Internal causes are due to configured resource limits or current limitations of Teralizer. External causes are due to Teralizer's dependencies (i.e., JUnit, Spoon, JPF / SPF, JaCoCo, and PIT). Mixed causes are influenced by both internal as well as external factors.

#	Type	Cause of Project-level Exclusion	Count
Stage 1 + 2 - Project Analysis:		*632 projects 130 inclusions 502 exclusions*	*20.6% inclusion rate*
1	Mixed	all assertions excluded due to filter rejections	255
2	Mixed	all tests excluded due to filter rejections and failures	129
3	Internal	timeout exceeded (60 seconds per Original test suite)	48
4	Internal	JUnit reports not found	31
5	Internal	compilation outputs not found	18
6	External	JUnit execution error during test execution	13
7	External	Spoon execution error during test analysis	8
Stage 3 - Specification Extraction:		*130 projects 117 inclusions 13 exclusions*	*90.0% inclusion rate*
8	Mixed	all assertions excluded due to earlier filter rejections and new failures	11
9	External	Spoon execution error during test instrumentation	1
10	Internal	timeout exceeded (60 seconds per Initial test suite)	1
Stage 4 - Generalized Test Creation:		*117 projects 114 inclusions 3 exclusions*	*97.4% inclusion rate*
11	Internal	all generalizations excluded due to filter rejections and failures	3
Stage 5 - Test Suite Reduction:		*114 projects 11 inclusions 103 exclusions*	*9.6% inclusion rate*
12	Internal	JaCoCo outputs not found	40
13	Internal	timeout exceeded (300 seconds per test suite variant)	40
14	External	PIT execution error during mutation testing	16
15	Internal	PIT reports not found	4
16	Internal	failed to process PIT reports	2
17	External	JaCoCo execution error during coverage collection	1
Overall:		*632 projects 11 inclusions 621 exclusions*	*1.7% inclusion rate*

The pipeline shows a distinct funnel pattern with two major barriers at the early and late processing stages, separated by high-success middle stages. Stage 1+2 (project analysis) excludes 79.4% of projects (502 of 632), forming the first major barrier. Projects that pass this initial filter progress largely successfully through Stage 3 (specification extraction, 90.0% pass rate: 117 of 130 projects) and Stage 4 (generalized test creation, 97.4% pass rate: 114 of 117 projects). The second major barrier emerges at Stage 5 (test suite reduction via mutation testing), where 90.4% of remaining projects are excluded (103 of 114 projects), leaving only 11 projects (1.7% of the original 632) to complete all stages. This pattern suggests that the core generalization mechanisms (Stages 3–4) operate reliably when projects match current tool capabilities, but both early filtering and final mutation testing highlight substantial practical challenges.

Stage 1 + 2 Exclusions: Project analysis excludes 502 of 632 projects (79.4%), with the primary failure mode being complete absence of suitable generalization candidates. Cases where all assertions are excluded affect 255 projects (40.3% of stage input), while all tests being excluded affects 129 projects (20.4%). For assertion exclusions, all 255 result from filter rejections, indicating that these projects contain only assertion patterns that are currently unsupported by Teralizer or beyond the current capabilities of the underlying symbolic analysis performed by SPF (detailed in Section 4.7.2). For test exclusions, 116 projects (89.9% of the 129) stem from filter rejections, 6 projects (4.7%) from failures during test analysis (e.g., missing test report files), and 7 projects (5.4%) from a combination of both.

Further internal exclusions caused by configured timeouts (60 seconds per project) and output detection failures affect 97 projects (15.3%). Timeout-based exclusions can be observed in 48 projects (7.6%) that have particularly long-running Original test suites. Output detection failures affect 49 projects (7.8%), split between failed JUnit report detection (31 projects, 4.9%) and failed compilation output detection (18 projects, 2.8%). Both types of output detection failures are caused by projects that store these outputs in non-standard output directories. Thus, these exclusions indicate that current search heuristics used by Teralizer cannot accommodate the full diversity of real-world project structures.

External execution errors affect 21 projects (3.3%): 13 projects (2.1%) encounter JUnit execution errors during test execution, and 8 projects (1.3%) encounter Spoon execution errors during test analysis. These failures stem from Teralizer's dependencies and, therefore, lie outside the direct control of Teralizer.

Stage 3 + 4 Exclusions: The 130 projects that complete project analysis face substantially lower exclusion rates in the two subsequent stages. This confirms that early filtering successfully identifies viable generalization candidates. Stage 3 (specification extraction) excludes 13 projects (10.0%) due to SPF execution errors, Teralizer errors, or exceeded resource limits (detailed in Section 4.7.2). One external failure represents a Spoon execution error during test instrumentation, and one internal failure occurs due to a test suite execution timeout (60 seconds per Initial test suite). Stage 4 (generalized test creation) shows the lowest failure rate in the pipeline: only 3 of 117 projects (2.6%) are excluded, all due to filter rejections that exclude all generalizations. The high inclusion rates at both stages (90.0% and 97.4% respectively) demonstrate that projects containing suitable assertions generally proceed successfully through specification extraction and test generation, supporting the pipeline's core generalization mechanisms.

Stage 5 Exclusions: Test suite reduction via mutation testing represents the second major exclusion barrier, excluding 103 of 114 projects that reach this stage (90.4%). Unlike Stage 1 + 2 failures that are primarily caused by proactive filtering of unsuitable tests and assertions, Stage 5 failures stem mainly from failures to detect required coverage and mutation testing reports, exceeded timeouts (300 seconds per test suite variant), and external tool failures.

In total, output detection failures affect 44 projects (38.6% of stage input). JaCoCo reports in non-standard locations cause 40 of these exclusions, while the remaining 4 projects are due to PIT reports in non-standard locations. These failures mirror the output detection issues seen in Stage 1 + 2, further supporting the observation that real-world projects organize build artifacts more diversely than TERALIZER's current output detection heuristics accommodate.

All 40 projects that are excluded due to exceeded runtime limits already reach the configured timeout (300 seconds per test suite variant) when performing mutation testing for the ORIGINAL test suite. No additional exclusions occur during mutation testing of the test suite created by the IMPROVED_{200} generalization strategy.

External execution errors affect 17 projects (14.9%): 16 projects encounter PIT errors during mutation testing and 1 project encounters a JaCoCo error during coverage collection. Two additional projects (1.8%) fail during PIT report parsing, where TERALIZER fails to successfully process the generated mutation reports.

4.7.2 *Test, Assertion, and Generalization Exclusions.*

Beyond project-level exclusions, individual tests, assertions, and generalizations are excluded throughout pipeline processing via filtering and due to processing failures. Table 13 quantifies exclusion rates across all 632 projects, distinguishing between filtering-based and failure-based exclusions. Table 14 provides further details about the causes of filter rejections across the three levels. For brevity, we only include the results for the IMPROVED_{200} generalization strategy. Since most exclusions occur in the SHARED processing stages, differences across generalization strategies are minor. Full results are available in our replication package [32].

Test-level Exclusions. Only 40.8% of real-world tests are included (33,385 of 81,810), compared to 83.1% under controlled conditions (Table 13 vs. Table 10). Of the 48,425 excluded tests, 49.6% are rejected through filtering while 9.6% are excluded due to processing failures. The high prevalence of filtering-based exclusions indicates that generalization of real-world tests commonly requires capabilities that are beyond what TERALIZER currently supports.

The `NoAssertions` filter shows the highest exclusion rate, rejecting 41.3% of real-world tests versus 10.3% under controlled conditions. However, RQ5 identified 86.3% of `NoAssertions` rejections in developer-written tests to be false positives: tests that are rejected by the `NoAssertions` filter but actually contain assertions in helper methods which TERALIZER's interprocedural static analysis does not detect. Given that RepoReapers exclusively contains developer-written tests, these rejections are likely to contain a high rate of false positives as well.

The `NonPassingTest` filter rejects 11.8% of real-world tests compared to the 6.6% rejection rate under controlled conditions. As explained in Section 4.6, this filter operates at the test class level because PIT requires a green test suite but only supports class-level exclusions. As a result, the filter rejects all 8,741 test methods from 974 classes containing at least one failing test. Of these, 4,709 (54%) actually failed during execution, while 4,032 (46%) passed but were rejected due to class-level filtering. Analysis of the 4,709 failing tests reveals that failures stem primarily from infrastructure and environment issues rather than broken tests: 18.8% encounter missing dependencies (NoClassDefFoundError), 14.1% fail due to unavailable external services (Redis, MongoDB, MySQL), 10.0% encounter null pointer exceptions, etc. Only 18.3% of failures are genuine assertion failures where tests execute to completion but produce incorrect results.

The `TestType` filter rejections increase from 0.8% under controlled conditions to 12.5% in real-world projects. Under controlled conditions, all rejections are @ParameterizedTest annotations in commons-utils-dev. In contrast, all 9,277 RepoReapers rejections are legacy JUnit 3 test methods that use the JUnit 3 naming convention (method names starting with "test") instead of @Test annotations, occurring across 70 different RepoReapers projects.

Assertion-level Exclusions. Assertion exclusions differ substantially between controlled and real-world conditions. Only 0.6% of assertions in RepoReapers projects are included (711 of 122,153) versus 47.8% under controlled conditions. Filtering accounts for 99.1% of exclusions while failures represent 0.3%, confirming that the primary barriers are known limitations of TERALIZER rather than unexpected failures during processing.

Table 13. Exclusion results for Improved$_{200}$ in the RepoReapers projects.

			Excluded	
Level	Total	Included	Filtering	Failures
Test	81,810	33,385 (40.8%)	40,583 (49.6%)	7,842 (9.6%)
Assertion	122,153	711 (0.6%)	121,060 (99.1%)	382 (0.3%)
Generalization	239	206 (86.2%)	23 (9.6%)	10 (4.2%)

Table 14. Filtering results for Improved$_{200}$ in the RepoReapers projects.

Level	Filter Name	Total	Accept	Defer	Reject
Test	NonPassingTest	74,308	65,567 (88.2%)	-	8,741 (11.8%)
Test	TestType	74,308	65,031 (87.5%)	-	9,277 (12.5%)
Test	NoAssertions	56,844	33,385 (58.7%)	-	23,459 (41.3%)
Assertion	AssertionType	122,153	92,986 (76.1%)	-	29,167 (23.9%)
Assertion	ExcludedTest	122,153	101,513 (83.1%)	-	20,640 (16.9%)
Assertion	MissingValue	122,153	51,425 (42.1%)	-	70,728 (57.9%)
Assertion	ParameterType	122,153	5,393 (4.4%)	56,477 (46.2%)	60,283 (49.4%)
Assertion	ReturnType	122,153	11,645 (9.5%)	70,728 (57.9%)	39,780 (32.6%)

The `MissingValue` filter shows the largest assertion-level rejection rate of 57.9%, compared to 24.7% under controlled conditions. Rejections have three underlying causes. First, 41% involve unsupported assertion types (assertThat, assertNull, fail, etc.) where method identification is not attempted. Second, 26% involve assertions where the actual value is not a method invocation or cannot be traced back to a method declaration, including field accesses, comparison expressions, and literal values. Third, 33% identify a method call but Spoon cannot resolve its declaration.

The `ParameterType` filter rejects 49.4% of assertions versus 15.4% under controlled conditions. This difference reflects dataset characteristics: the controlled dataset used methods with primarily numeric and boolean parameters. In contrast, real-world projects show 52.6% no-parameter methods, 26.9% object and array parameters, and only 19.8% numeric and boolean parameters. No-parameter methods cannot benefit from input generalization, while methods with object and array parameters require capabilities beyond Teralizer's current support for numeric and boolean types.

The `ReturnType` filter defers on 57.9% of assertions where the method is unknown (matching the `MissingValue` rejection rate) and rejects 32.6% of all assertions. Among methods with known return types (42.1% of all assertions), 52.7% return objects, 46.6% return numeric or boolean types, and 0.7% return other types (char, arrays, void). This contrasts with controlled conditions where 98.3% of methods returned numeric and boolean types.

The `AssertionType` filter rejects 23.9% of real-world assertions versus 2.6% under controlled conditions. This increase stems from more diverse assertion usage in real-world test code. For example, assertEquals accounts for 83.6% of assertions in controlled conditions, primarily due to the large number of EvoSuite-generated tests which use assertEquals in 84.7% of cases. In contrast, assertEquals accounts for only 54.5% of assertions in the RepoReapers projects. The most common unsupported assertion types are assertThat (8.3%), assertNotNull (6.1%), fail (3.3%), and assertNull (3.1%). In controlled conditions, these four types collectively account for only 0.8% of assertions.

The `ExcludedTest` filter shows a 3× increase from 5.5% under controlled conditions to 16.9% in real-world projects, reflecting the corresponding increase in test-level exclusions from 16.9% to 59.2%.

Generalization-level Exclusions. Of 239 generalization attempts in the RepoReapers projects, 206 (86.2%) succeed. The 33 exclusions result from `NonPassingTest` filter rejections (23 cases, 9.6%) and test report detection failures (10 cases, 4.2%). The 86.2% inclusion rate is comparable to the 83.8% rate under controlled conditions. However, only 0.2% of all assertions result in an included generalization (206 of 122,153), compared to 40.1% under controlled conditions (11,597 of 28,923). These results demonstrate that the core generalization mechanism operates reliably when applicable, but real-world applicability is limited by three primary barriers: limited type and assertion pattern support (99.4% of assertions excluded by filters at earlier stages), non-standard project structures (output detection failures exclude 14.7% of projects at Stages 1+2 and 5), and resource constraints (timeout exclusions affect 14.1% of projects across all stages).

Answer to RQ6: Fully automated generalization of real-world test suites encounters significant challenges. Only 206 of 122,153 assertions (0.2%) are successfully generalized (compared to 40.1% under controlled conditions) and only 11 of 632 projects (1.7%) complete all processing stages. The core generalization mechanism operates reliably when applicable: generalization-level success rates are comparable (86.2% real-world vs 83.8% controlled), and projects that pass early filtering also complete specification extraction (90.0%) and generalized test creation (97.4%). However, three barriers prevent higher overall success rates: limited type and assertion support causes 99.4% of assertions to be filtered (e.g., `MissingValue`: 57.9%, `ParameterType`: 49.4%, `ReturnType`: 32.6%), non-standard project structures prevent output detection in 14.7% of projects, and execution timeouts exclude 14.1% of projects.

5 Discussion

Our evaluation shows that semantics-based test generalization via symbolic analysis is viable but currently constrained to specific application environments and test architectures. Under controlled conditions that match current capabilities, Teralizer successfully generalized 40.1% of assertions (RQ5) and improved mutation detection by 1–4 percentage points (RQ1). As post-processing for generated tests, generalization offers competitive efficiency: combining 1-second EvoSuite generation with Teralizer's generalization achieved comparable mutation detection to 60-second generation while reducing processing time by 31.9% (RQ4). Generalization of real-world projects faces substantial barriers (RQ6): only 0.6% of assertions passed analysis and filtering stages (versus 47.8% under controlled conditions), only 0.2% of assertions successfully generalized, and only 1.7% of real-world projects completed the processing pipeline. This section discusses when and why generalization succeeds (Section 5.1), when and why it fails (Section 5.2), and how future research and engineering efforts can improve generalization effectiveness, efficiency, and applicability (Section 5.3).

5.1 When and Why Generalization Succeeds

Generalization succeeds when implementation and test properties of the target projects align with Teralizer's current static analysis capabilities as well as the capabilities of SPF's symbolic analysis. Implementation code that is amenable to generalization is primarily focused on numeric computations in pure deterministic functions without any side effects (thus enabling symbolic analysis) and is organized in standard project structures that facilitate detection of required output artifacts such as compilation outputs as well as (mutation) testing and coverage reports. Tests amenable to generalization are single-assertion unit tests without complex setup logic, loops, or interprocedural control flow.

When these conditions are satisfied, Teralizer achieves moderate mutation score improvements at reasonable runtime cost. Mutation scores increased by 1.2–3.9 percentage points in eqbench-es-∗ and by 0.82–1.33 percentage points in commons-utils-es-∗ compared to EvoSuite-generated baselines (Figure 4). Beyond effectiveness, generalization offers competitive efficiency when combined with test generation. For example, 1-second EvoSuite generation combined with Naive$_{50}$ generalization achieves 51.7% mutation detection rate in 37,532 seconds, thus outperforming 60-second generation alone which achieves 51.6% mutation detection rate in 55,075 seconds (Figure 7).

Outcomes vary based on constraint complexity and original test suite effectiveness of the target projects. More complex constraints hinder mutation score improvements because they increase the number of `TooManyFilterMisses-Exceptions`. Similarly, stronger original test suites leave less room for mutation score improvements. For example, Naive and Improved both show larger mutation score improvements for eqbench-es-$*$ than for commons-utils-es-$*$ because of the simpler constraints in eqbench-es-$*$ (137–231 vs. 290–507 average operation counts, Table 4), and larger mutation score improvements for commons-utils-es-$*$ than for commons-utils-dev because of commons-utils-dev's stronger original tests (56.77–58.12% vs. 80.35% Initial mutation detection rate, Figure 4).

Naive is more effective than Improved for simpler constraints (Figure 4, rows 1–3), whereas Improved is more effective for more complex constraints (Figure 4, rows 4–6). Two factors explain these results (Section 4.3). First, simpler constraints are easier to satisfy by chance. Consequently, the difference in `TooManyFilterMissesExceptions` between Naive and Improved is smaller in such cases than for more complex constraints. Second, simpler constraints enable Improved to more reliably encode input partition boundaries. As a result, it spends more `tries` on boundary testing but neglects non-boundary testing, which limits mutation detection improvements. This effect is more pronounced at low `tries` settings where Improved$_{10}$ underperforms all other generalization strategies (Figure 4, rows 1–3).

5.2 When and Why Generalization Fails

As shown by RQ6, generalization failed for the vast majority of evaluated real-world projects (Section 4.7). We identify three high-level causes that explain these high exclusion rates. First, Teralizer is a research prototype, which limits its current capabilities. Second, extracting accurate specifications for generalization of test oracles is a non-trivial problem, even more so when moving beyond the domain of pure functions and numerical programs. Third, factors such as execution errors in Teralizer's dependencies, timeouts enforced for evaluation purposes, and test failures in original test suites are beyond the direct control of the generalization mechanism itself, but still increase the number of unsuccessful generalization attempts. The following subsections describe how each of these causes affects generalization outcomes. This summary of failure causes then serves as the basis for the discussion of future improvements in Section 5.3.

Implementation Limitations. There are four limiting factors in the current implementation of our prototype: (i) it only supports JUnit 4 and JUnit 5 tests and assertions, (ii) it only supports generalization of tests that contain at least one assertion, (iii) it only performs intraprocedural static analysis within test methods to detect assertions, and (iv) it only supports projects that use default output directories for compilation outputs and test reports. Limitation (iv) directly causes 95 project-level exclusions (15.0% of projects) due to output detection and processing failures (Table 12, rows #4, #5, #12, #15, and #16). Limitations (i)–(iii) contribute to the exclusion of 129 projects (20.4%) for which all tests are excluded (Table 12, row #2) and 266 projects (42.1%) for which all assertions are excluded (Table 12, rows #1 and #8).

While the exact impact on the 129 test- and 266 assertion-related exclusions is difficult to quantify precisely (because both are also affected by the other two high-level factors), filter rejections provide at least an approximate measure. Limitation (i) causes all 12.5% of `TestType` rejections (Table 14) because these tests use JUnit 3. Furthermore, limitation (ii) causes all true positive `NoAssertions` rejections and (i)+(iii) cause all false positive `NoAssertions` rejections, together accounting for the exclusion of 41.3% of tests (Table 14). Thus, limitations (i)–(iii) have comparatively high impact on the 129 test-related exclusions — the only other test-excluding factor is 11.8% `NonPassingTest` rejections.

In contrast, their impact on the 266 assertion-related exclusions is comparatively low, contributing only to 16.9% `ExcludedTest` rejections — the lowest rate among all assertion-level filters (Table 14).

Specification Extraction Challenges. Whereas implementation limitations primarily cause direct project-level and test-related exclusions, specification extraction challenges account for the majority of the 266 assertion-related exclusions (42.1% of projects) as well as all 3 (0.5%) generalization-related exclusions (Table 12, rows #1, #8, and #11). The largest portion of these exclusions are due to type limitations of the underlying symbolic analysis performed by SPF (discussed in Section 2.3), which is used by Teralizer to extract specifications for oracle generalization (Sections 3.3 and 3.4). A smaller portion is due to the assumption that each assertion can be generalized by extracting the input-output specification of the last method call that was executed before the assertion (Section 3.2).

To quantify the impact of type limitations, notice that they are responsible for the following assertion filter rejections listed in Table 14: all `ParameterType` rejections (49.4% rejection rate), all `ReturnType` rejections (32.6%), most `AssertionType` rejections (23.9%), and many `MissingValue` rejections (57.9%). `AssertionType` is type-related because many unsupported assertions are for non-primitive types (`assert(Not)Null`, `assert(Not)Same`, `assertArrayEquals`, etc.). `MissingValue` rejections are type-related because they are a superset of `AssertionType` rejections. `ParameterType` and `ReturnType` rejections are directly enforced due to type limitations. Furthermore, the exclusion rates relative to the subset of cases for which type information is available are even higher than overall rejection rates suggest. Of 65,676 cases with parameter type information, 60,283 (91.8%) are rejected by the `ParameterType` filter. Similarly, 39,780 of 51,425 cases (77.4%) with return type information are rejected by the `ReturnType` filter.

Exclusions due to the 1:1 assertion-to-MUT mapping assumption show a smaller impact on overall exclusion rates. Specifically, this assumption causes the subset of `MissingValue` rejections that occur when no MUT can be identified for a given assertion. As explained in Section 4.7.2, this subset accounts for 26% of `MissingValue` rejections, which in turn reject 57.9% of identified assertions — thus contributing rejection votes for approximately 15% of all assertions. However, this figure likely understates the assumption's true impact: since assertion-to-MUT mapping is only attempted for supported assertions, other rejection causes such as type limitations shadow an unknown portion of mapping failures that would be revealed if type and assertion support were improved.

Dependencies and Environment. Generalization success is also affected by execution errors in Teralizer's dependencies and environmental factors such as resource limits. These factors are beyond the control of the generalization approach but account for 128 direct project-level exclusions (20.3% of projects). Furthermore, they contribute to 129 test-related exclusions (20.4%) and 266 assertion-related exclusions (42.1%). The 128 project-level exclusions are caused by 89 timeouts (Table 12, rows #3, #10, and #13) and 39 dependency errors (Table 12, rows #6, #7, #9, #12, and #14). The 129 test-related exclusions are affected by `NonPassingTest` rejections (11.8% of tests, Table 14), and the 266 assertion-related exclusions are affected by `ExcludedTest` rejections (16.9% of assertions), `MissingValue` rejections (57.9%, 33% of which are cases where Spoon is unable to resolve the declaration of an identified MUT, see Section 4.7.2), and SPF execution failures (0.3%). SPF failures are underrepresented because most assertions are excluded before specification extraction. Among the 1093 assertions that reach SPF, 382 (34.9%) fail due to SPF errors and enforced resource limits (Table 13).

5.3 Directions for Future Improvements

Based on our findings, three improvement directions emerge for semantics-based test generalization: (i) expanding applicability to handle more projects, tests, and assertions, (ii) improving effectiveness when generalization does apply, and (iii) improving efficiency in terms of generalization runtime as well as size and runtime of generalized test suites.

This section discusses opportunities in each direction, distinguishing between engineering improvements achievable through additional implementation effort and research challenges requiring advances in underlying techniques.

5.3.1 Improving Applicability. The primary barrier to real-world adoption is limited applicability: 99.4% of real-world assertions are excluded before reaching generalized test creation (Table 13). Three categories of improvements could noticeably expand the subset of projects, tests, and assertions that are amenable to generalization.

Type Support. Type limitations cause the largest portion of assertion-level exclusions. Among assertions where type information is available, type-based rejection rates reach 91.8% for cases with known parameter types and 77.4% for cases with known return types (Section 5.2). Expanding type support remains a fundamental research challenge [1, 13, 14, 73, 76]: precise constraint modeling for strings, arrays, and objects requires advances in symbolic analysis that go beyond the capabilities of current tools and approaches. Furthermore, type limitations shadow other issues. Thus, as type support improves, additional limitations in assertion-to-MUT mapping and general assertion support would become visible, enabling more in-depth analysis and targeted improvement of these causes.

Static Analysis. The `NoAssertions` and `TestType` filters reject 41.3% and 12.5% of tests, respectively (Section 5.2). Both are addressable through engineering improvements: interprocedural analysis that tracks assertion calls through the call graph would recover tests where assertions exist in helper methods. Tests that genuinely lack assertions could be modeled as implicit "does not throw" checks, thus eliminating the current need to exclude tests without assertions due to a lack of validated oracles. Adding support for JUnit 3, TestNG, and assertion libraries such as AssertJ, Hamcrest, and Truth would recover further rejections by the `TestType` and `NoAssertions` filters.

Project Structure and Environment. Output detection failures exclude 14.7% of projects (Table 12). Configurable output paths or improved search heuristics would recover these projects without changing the core approach. Timeouts exclude 14.1% of projects across processing stages (Table 12). While increased limits could reduce these exclusions, diminishing returns are apparent: doubling of all timeouts recovered only 2 of 89 timed-out projects in our internal testing, increasing the number of successfully processed real-world projects from 11 to 13. SPF execution errors account for 51.4% of specification extraction failures under controlled settings (Section 4.6), many due to missing models for native methods. Contributing such models to SPF would reduce these failures without any other changes in the approach.

5.3.2 Improving Effectiveness. When generalization does apply, effectiveness depends on the generation of inputs that thoroughly cover the valid input space. Two factors influence this: (i) the generation strategy, which determines whether inputs are sampled randomly (Naive) or specifically target input partition boundaries (Improved), and (ii) constraint encoding, which determines how much of the extracted specification can be used to guide generation.

Generation Strategies. Constraint-aware input generation used by Improved increases detection rate improvements of boundary-related mutations such as `ConditionalsBoundary` compared to Naive (+2.55pp vs. +1.21pp, Section 4.2.2). However, focusing too much on boundaries limits arithmetic diversity within available `tries`. This negatively affects detection rate improvements of `Math` mutations, particularly at lower `tries` settings (Section 4.3). To improve detection rates without increasing `tries`, more balanced generation strategies can be developed that use heuristics based on constraint complexity or other source code properties to better balance boundary vs. non-boundary testing, thus utilizing the benefits of both random and boundary-focused generation where each provides the largest benefit.

Constraint Encoding. Average constraint utilization per project ranges from 11% to 69% in controlled settings (Table 4). This is because Teralizer's Improved generalization strategy only encodes simple in-/equalities on variables and constants, whereas compound terms such as `a == (b + 1)` are enforced through filtering (Section 3.4.3). Extending constraint encoding support would enable more precise boundary testing while reducing `TooManyFilterMissesExceptions` that affect 5.7–15.7% of generalizations across datasets (Section 4.6). However, as constraint complexity increases, generating valid inputs becomes increasingly difficult. Techniques such as SMT-based constraint solving [23, 70], targeted property-based testing [45], or coverage-guided property-based testing [43] could more effectively generate inputs satisfying complex constraints than jqwik's primarily random generation, albeit at increased computational cost.

5.3.3 *Improving Efficiency.* Efficiency improvements could be implemented along the following three dimensions: (i) tool runtime, which determines processing cost during generalization, (ii) test suite runtime, which determines execution cost after generalization, and (iii) test suite size, which primarily affects maintenance overhead.

Tool Runtime. Processing costs are dominated by mutation testing, which consumes 59.1–95.7% of total pipeline runtime (Figure 6). However, not all mutation operators benefit equally from generalization (Table 3). Focusing on operators that benefit the most from generalization and using lightweight heuristics based on constraint complexity or assertion patterns to identify unlikely-beneficial candidates before mutation testing could reduce processing time at the cost of potentially missing some improvements. Incremental processing that targets only newly-added or modified tests would avoid repeated analysis of stable code in continuous integration settings. Arcmutate [82], a commercial extension of PIT, offers an accelerator plugin that could further reduce processing costs.

Test Suite Runtime. Test suite runtime increases caused by generalization stem primarily from jqwik framework overhead (approximately 150ms per test, Figure 5) and filter-and-regenerate cycles when inputs violate constraints. Improved reduces filter miss rates by 46.5–58.4% compared to Naive (Section 4.6), which reduces execution cost per successfully generated test input from 28.61ms to 18.92ms at 10 `tries` and from 5.68ms to 1.98ms at 200 `tries` (Figure 5). Better constraint encoding would further reduce filter-and-regenerate cycles by enabling more inputs to be generated directly rather than requiring filtering. Additionally, jqwik 2 plans parallelization support [44], which could reduce test suite execution time by distributing property-based test execution across multiple cores.

Test Suite Size. Observed LOC increases of 4.9–58.7% across projects (Table 6) have two primary causes: explicit constraint encoding in generalized tests and structural duplication from test isolation (Section 4.4.2). Abstracting constraint encoding in a library could reduce this overhead, and tighter integration of generalized tests into original test classes would avoid duplication from copied imports and helper methods. Test suite reduction could also be extended to replace multiple original tests that cover the same partition with a single property-based test, thus reducing test count instead of only compensating for added tests. This mirrors the idea of test suite reduction via parameterization [3, 83], but would use semantics-based analysis rather than syntactic clone detection to identify mergeable tests.

5.3.4 *Deployment Strategies.* Our results position semantics-based test generalization for targeted deployment during new unit test development or as a post-processing step for generated tests in numeric-heavy domains. Because generalization success depends not only on domain characteristics but also on program and test architecture, developers who are interested in adopting automated test generalization tools such as Teralizer can improve generalization outcomes through their implementation choices independent of further test generalization advances:

(1) Following a more functional programming style that emphasizes pure functions reduces assertion-to-MUT mapping failures (57.9% `MissingValue` rejections, Table 14).
(2) Placing assertions directly in test methods rather than delegating to helper methods reduces assertion detection failures due to current interprocedural analysis (41.3% `NoAssertions` rejections, Table 14).
(3) Favoring supported assertions such as `assertEquals` over unsupported ones such as `assertThat` where this is feasible reduces assertion type exclusions (23.9% `AssertionType` rejections, Table 14).

(4) Ensuring a green original test suite by addressing flaky tests and missing dependencies avoids exclusions due to failing tests (11.8% `NonPassingTest` rejections, Table 14).
(5) Using standard project structures and build output locations for test reports and coverage data reduces output detection failures (14.7% project exclusions, Table 12).

Beyond these factors, test smells [31, 51, 84] represent another dimension that affects generalizability. For example, *Eager Test* (where tests invoke multiple production methods) complicates assertion-to-MUT mapping (Section 3.2) and *Conditional Test Logic* (where tests contain loops or other conditionals) can lead to inaccurate specifications due to Teralizer's limited loop handling (Section 4.6). Thus, another direction for future work is developing automated transformation approaches that refactor test code to improve generalizability before applying tools like Teralizer. Such transformations could build on testability transformation techniques [35], which modify programs to improve amenability to test generation, and recent advances in automated test smell detection [67] and refactoring [49, 85].

5.4 Threats to Validity

Construct Validity. We use mutation score as a proxy for fault detection capability. Fundamentally, this use of mutation testing rests on two hypotheses: the competent programmer hypothesis, which assumes that real faults are often only small deviations from correct programs, and the coupling effect, which suggests that tests which detect simple faults will also generally detect more complex faults [24]. Empirical studies validated the coupling effect [59], and subsequent work demonstrated that mutation scores correlate with real fault detection [42, 63]. Furthermore, surveys confirm the use of mutation testing as a standard evaluation technique in software testing research [40, 62]. While our use of PIT's `DEFAULTS` set of mutation operators may not represent all fault types, `DEFAULTS` is explicitly recommended by PIT as a stable set of operators that minimizes equivalent mutants and avoids subsumption [16, 17].

Internal Validity. Our experiments use single runs per configuration. While additional runs would produce more robust results, we already observe consistent effectiveness and efficiency trends across projects and configuration settings with our current setup. Similarly, evaluation of higher `tries` and longer timeouts could provide further evidence of scaling behaviors. However, we empirically determined these settings to provide a reasonable trade-off between resource requirements and result quality. Scaling trends and diminishing returns are already apparent throughout the evaluation, and doubled timeout settings recovered only 2 of 89 timed-out projects in our internal testing.

External Validity. Our implementation targets Java 5–8 with JUnit 4/5 and Maven or Gradle builds. While the core approach is programming language-agnostic and could be implemented for other languages and ecosystems (e.g., using KLEE [11] with RapidCheck [26] for C/C++, or CrossHair [72] with Hypothesis [48] for Python), observed results might differ due to language differences and maturity of available tools. Our evaluation of benefits emphasizes projects that match current symbolic analysis capabilities, particularly regarding type support limitations. As type support improves, applicability of semantics-based test generalization would directly benefit, but results that can be achieved for non-numeric types might differ from those we observed during generalization of primarily numerical programs.

6 Related Work

Our work draws on ideas from test amplification and symbolic analysis to automate the transformation from conventional unit tests to property-based tests. This section reviews prior approaches to test generalization (Section 6.1), discusses research approaches and directions that could improve specification inference capabilities of our current prototype (Section 6.2), and explores synergies with related techniques as well as developer perspectives (Section 6.3).

6.1 Test Generalization

Property-based testing [15] and parameterized unit testing [79] enable multi-input validation through general properties, differing primarily in input generation strategy: property-based tests (PBTs) traditionally use random generation to produce inputs, whereas Tillmann and de Halleux suggest to execute parameterized unit tests (PUTs) symbolically, utilizing constraint solving to select inputs for test parameters [78]. Both approaches require developers to manually specify general assertions that hold across ranges of inputs rather than specific input-output examples used in conventional unit tests (CUTs). Thummalapenta et al. [77] demonstrated manual strategies for retrofitting CUTs to PUTs.

Fraser and Zeller [30] automated generation of PUTs from CUTs, but use tests without existing assertions as a starting point. This sidesteps the problem of automated oracle generalization. However, it often causes generated PUTs to overfit the implementation [30] because the lack of validated oracles makes it difficult to distinguish intentional behavior from incidental state changes or outputs. PROZE [80] uses runtime inputs and outputs to transform CUTs to PUTs but does not generalize beyond observed values. JARVIS [66] introduced automated CUT-to-PBT transformation using black-box analysis with predefined abstraction templates, which produces overapproximations that require multiple related tests to constrain. We instead use white-box symbolic analysis along concrete execution paths, extracting path-exact specifications that generalize oracles from individual input-output examples.

6.2 Specification Inference

Type support limitations fundamentally constrain specification extraction through single-path symbolic analysis, as discussed in Sections 3.3, 4.7, and 5.2. These limitations stem from our reliance on SPF [64], a symbolic execution tool designed for path exploration. Because full symbolic execution requires constraint solving to determine path feasibility, SPF only encodes constraints for types with adequate solver support. Extending solver capabilities remains an active research area, with recent work showing improvements for string constraints [13, 34, 46], heap-allocated structures [8, 19, 20], arrays [57], and floating-point arithmetic [86]. As solver support improves and symbolic execution tools correspondingly extend their constraint encoding, semantics-based test generalization would also benefit.

Alternative approaches to specification inference largely avoid type support limitations inherent to symbolic analysis, but infer general specifications that describe overall method behavior rather than path-exact constraints, which complicates oracle generalization. Houdini [28] pioneered template-based inference, generating candidate annotations and using verification to filter them. Daikon [27] introduced dynamic invariant detection from execution traces. More recent tools target specific specification types: EvoSpex [53] uses evolutionary search to infer postconditions, SpecFuzzer [52] combines grammar-based fuzzing with mutation analysis for class specifications, and PreCA [50] employs constraint acquisition [7] to infer preconditions from input-output observations. LLM-based techniques offer yet another path: SpecGen [47] uses conversational prompting with mutation-based refinement to generate specifications from source code, whereas ClassInvGen [75] co-evolves class invariants with test inputs.

6.3 Test Generation and Developer Perspective

Test generalization builds on existing tests and their assertions, creating natural synergies with techniques that produce or enrich them. Test generation tools such as EvoSuite [29], Randoop [61], and UTBot [81] produce complete unit tests through search-based, random, and hybrid approaches, while DSpot [22] amplifies existing tests to cover additional branches. Oracle inference techniques such as TOGA [25], TOGLL [36], and AsserT5 [69] add assertions to tests that lack them. All of these expand the pool of available generalization candidates. RQ4 demonstrates this combination:

pairing EvoSuite's generation with Teralizer's generalization achieves higher mutation scores at lower runtimes than test generation alone. However, generated tests and inferred oracles risk overfitting the implementation rather than capturing intended specifications [6], and this risk carries through to any subsequent generalization.

For use cases beyond fully automated pipelines, developer interaction with generalized tests becomes relevant. Studies of test amplification show that developers filter and edit amplified tests extensively before adding them to their test suites [9, 10]. By making minimal structural changes — parameterizing inputs and expected values while preserving the original test logic — test generalization may reduce friction compared to approaches that generate entirely new test code. However, property-based testing introduces its own complexity: moving from example-based to property-based thinking requires a conceptual shift that can be difficult for developers [33, 38]. Thus, generator constraints and generalized oracles that replace concrete values must be presented appropriately for developers to understand and trust them. Improving understandability of generalized tests therefore represents a research direction that should be tackled in future work to better support use cases outside of fully automated testing scenarios.

7 Conclusions

This paper introduced a semantics-based approach for automated test generalization, using specifications extracted through single-path symbolic analysis to transform conventional unit tests into property-based tests. We implemented this approach in a prototype tool called Teralizer. Under controlled conditions matching current symbolic analysis capabilities, Teralizer achieves mutation score improvements of 1–4 percentage points compared to EvoSuite-generated baselines. Pareto analysis further showed that combining short test generation with test generalization can outperform longer generation alone. For example, 1-second generation plus generalization achieves a higher mutation score on EqBench than 60-second generation (51.7% vs 51.6%) while requiring 32% less total runtime.

However, our evaluation across 632 real-world Java projects from the RepoReapers dataset reveals substantial barriers to fully automated generalization under real-world conditions: only 1.7% of projects complete the processing pipeline, and 98.3% of assertions are excluded before reaching generalized test creation. By analyzing these exclusions in detail, we distinguish implementation limitations of our prototype from fundamental research challenges in specification extraction, providing concrete guidance for advancing the field. The primary barrier to fully automated generalization is limited type support in existing symbolic analysis tools and approaches: current tools cannot precisely encode constraints for strings, arrays, and objects, causing the majority of assertion-level exclusions.

As symbolic analysis improves to support additional types, semantics-based test generalization would directly benefit. Other limitations of Teralizer are addressable through engineering improvements without requiring research advances: interprocedural analysis would recover assertions in helper methods, broader framework support would reduce test-level exclusions, and extended constraint encoding in generated tests would improve effectiveness and efficiency by reducing filter-and-regenerate cycles. Our complete implementation and replication package are publicly available to support reproduction and extension of this work [32].

Acknowledgments

This research was funded in whole or in part by the Austrian Science Fund (FWF) 10.55776/P36698. For open access purposes, the author has applied a CC BY public copyright license to any author accepted manuscript version arising from this submission. The research was supported by the Austrian ministries BMIMI, BMWET and the State of Upper Austria in the frame of the SCCH COMET competence center INTEGRATE (FFG 892418).

References

[1] Roberto Amadini. 2023. A Survey on String Constraint Solving. *Comput. Surveys* 55, 2 (2023), 16:1–16:38. doi:10.1145/3484198

[2] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing* (2nd ed.). Cambridge University Press.

[3] Aidin Azamnouri and Samad Paydar. 2021. Compressing Automatically Generated Unit Test Suites Through Test Parameterization. In *Proceedings of the 9th International Conference on Fundamentals of Software Engineering*. Springer, 215–221. doi:10.1007/978-3-030-89247-0_15

[4] Sahar Badihi, Yi Li, and Julia Rubin. 2021. EqBench: A Dataset of Equivalent and Non-Equivalent Program Pairs. In *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories*. IEEE, 610–614. doi:10.1109/msr52588.2021.00084

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (2018), 50:1–50:39. doi:10.1145/3182657

[6] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/tse.2014.2372785

[7] Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. 2017. Constraint Acquisition. *Artificial Intelligence* 244 (2017), 315–342. doi:10.1016/J.ARTINT.2015.08.001

[8] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2016. JBSE: A Symbolic Executor for Java Programs with Complex Heap Inputs. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 1018–1022. doi:10.1145/2950290.2983940

[9] Carolin Brandt, Ali Khatami, Mairieli Wessel, and Andy Zaidman. 2024. Shaken, Not Stirred: How Developers Like Their Amplified Tests. *IEEE Transactions on Software Engineering* 50, 5 (2024), 1264–1280. doi:10.1109/tse.2024.3381015

[10] Carolin Brandt and Andy Zaidman. 2022. Developer-Centric Test Amplification. *Empirical Software Engineering* 27, 4 (2022), 96. doi:10.1007/S10664-021-10094-2

[11] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[12] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90. doi:10.1145/2408776.2408795

[13] Yu-Fang Chen, David Chocholatý, Vojtech Havlena, Lukás Holík, Ondrej Lengál, and Juraj Síc. 2024. Z3-Noodler: An Automata-based String Solver. In *Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 24–33. doi:10.1007/978-3-031-57246-3_2

[14] David Chocholatý, Vojtech Havlena, Lukás Holík, Jan Hranicka, Ondrej Lengál, and Juraj Síc. 2025. Z3-Noodler 1.3: Shepherding Decision Procedures for Strings with Model Generation. In *Proceedings of the 31st International Conferenece on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 23–44. doi:10.1007/978-3-031-90653-4_2

[15] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*. ACM, 268–279. doi:10.1145/351240.351266

[16] Henry Coles. [n. d.]. *Pitest - Mutation Operators*. https://pitest.org/quickstart/mutators/ Accessed: 2025-12-10.

[17] Henry Coles. 2021. *Pitest Blog - Less is More*. https://blog.pitest.org/less-is-more/ Accessed: 2025-12-10.

[18] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 449–452. doi:10.1145/2931037.2948707

[19] Juan Manuel Copia, Facundo Molina, Nazareno Aguirre, Marcelo F. Frias, Alessandra Gorla, and Pablo Ponzio. 2023. Precise Lazy Initialization for Programs with Complex Heap Inputs. In *Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering*. IEEE, 752–762. doi:10.1109/issre59848.2023.00080

[20] Juan Manuel Copia, Pablo Ponzio, Nazareno Aguirre, Alessandra Gorla, and Marcelo F. Frias. 2022. LISSA: Lazy Initialization with Specialized Solver Aid. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 67:1–67:12. doi:10.1145/3551349.3556965

[21] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. 2019. A Snowballing Literature Study on Test Amplification. *Journal of Systems and Software* 157 (2019). doi:10.1016/J.JSS.2019.110398

[22] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. 2019. Automatic Test Improvement with DSpot: A Study with Ten Mature Open-Source Projects. *Empirical Software Engineering* 24, 4 (2019), 2603–2635. doi:10.1007/S10664-019-09692-Y

[23] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[24] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. doi:10.1109/C-M.1978.218136

[25] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering*. ACM, 2130–2141. doi:10.1145/3510003.3510141

[26] Emil Eriksson. 2025. *RapidCheck: QuickCheck Clone for C++*. https://github.com/emil-e/rapidcheck Accessed: 2025-12-10.

[27] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming* 69, 1-3 (2007), 35–45. doi:10.1016/J.SCICO.2007.01.015

[28] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe*. Springer, 500–517. doi:10.1007/3-540-45251-6_29

[29] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and the 13th European Software Engineering Conference*. ACM, 416–419. doi:10.1145/2025113.2025179

[30] Gordon Fraser and Andreas Zeller. 2011. Generating Parameterized Unit Tests. In *Proceedings of the 20th International Symposium on Software Testing and Analysis*. ACM, 364–374. doi:10.1145/2001420.2001464

[31] Vahid Garousi and Baris Küçük. 2018. Smells in Software Test Code: A Survey of Knowledge in Industry and Academia. *Journal of Systems and Software* 138 (2018), 52–81. doi:10.1016/J.JSS.2017.12.013

[32] Johann Glock, Clemens Bauer, and Martin Pinzger. 2025. *Replication Package for: "Teralizer: Automated Test Generalization from Conventional Unit Tests to Property-Based Tests"*. doi:10.5281/zenodo.17950380

[33] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. ACM, 187:1–187:13. doi:10.1145/3597503.3639581

[34] Matthew Hague, Denghang Hu, Artur Jeż, Anthony W. Lin, Oliver Markgraf, Philipp Rümmer, and Zhilin Wu. 2025. OSTRICH2: Solver for Complex String Constraints. arXiv:2506.14363 [cs.LO] https://arxiv.org/abs/2506.14363

[35] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability Transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3–16. doi:10.1109/tse.2004.1265732

[36] Soneya Binta Hossain and Matthew B. Dwyer. 2025. TOGLL: Correct and Strong Test Oracle Generation with LLMS. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*. IEEE, 1475–1487. doi:10.1109/icse55347.2025.00098

[37] John Hughes. 2007. QuickCheck Testing for Fun and Profit. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*. Springer, 1–32. doi:10.1007/978-3-540-69611-7_1

[38] John Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer, 169–186. doi:10.1007/978-3-319-30936-1_9

[39] Laura Inozemtseva and Reid Holmes. 2014. Coverage Is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 435–445. doi:10.1145/2568225.2568271

[40] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/tse.2010.62

[41] jqwik-team. [n. d.]. *jqwik: Property-Based Testing in Java*. https://jqwik.net/ Accessed: 2025-12-10.

[42] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 654–665. doi:10.1145/2635868.2635929

[43] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 181:1–181:29. doi:10.1145/3360607

[44] Johannes Link. 2024. *The Next Generation of Property-Based Testing*. https://blog.johanneslink.net/2024/09/13/next-gen-pbt/ Accessed: 2025-12-10.

[45] Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 46–56. doi:10.1145/3092703.3092711

[46] Kevin Lotz, Mitja Kulczynski, and Dirk Nowotka. 2025. S2S: An Eager SMT Solver for Strings. In *Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design*. TU Wien Academic Press, 133–138. doi:10.34727/2025/isbn.978-3-85448-084-6_19

[47] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*. IEEE, 16–28. doi:10.1109/icse55347.2025.00129

[48] David Maciver and Zac Hatfield-Dodds. 2019. Hypothesis: A New Approach to Property-Based Testing. *Journal of Open Source Software* 4, 43 (2019), 1891. doi:10.21105/JOSS.01891

[49] Luana Almeida Martins, Taher Ahmed Ghaleb, Heitor A. X. Costa, and Ivan Machado. 2024. A Comprehensive Catalog of Refactoring Strategies to Handle Test Smells in Java-based Systems. *Software Quality Journal* 32, 2 (2024), 641–679. doi:10.1007/S11219-024-09663-7

[50] Grégoire Menguy, Sébastien Bardin, Arnaud Gotlieb, and Nadjib Lazaar. 2025. A Query-Based Constraint Acquisition Approach for Enhanced Precision in Program Precondition Inference. *Journal of Artificial Intelligence Research* 82 (2025), 901–936. doi:10.1613/JAIR.1.16206

[51] Gerard Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

[52] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2023. SpecFuzzer: A Tool for Inferring Class Specifications via Grammar-Based Fuzzing. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2094–2097. doi:10.1109/ase56229.2023.00024

[53] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2023. EvoSpex: A Search-Based Tool for Postcondition Inference. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 1519–1522. doi:10.1145/3597926.3604928

[54] Mountainminds GmbH & Co. KG and Contributors. [n. d.]. *JaCoCo Java Code Coverage Library*. https://www.jacoco.org/jacoco/ Accessed: 2025-12-10.

[55] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for Engineered Software Projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253. doi:10.1007/S10664-017-9512-6

[56] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). John Wiley & Sons.

[57] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *Proceedings of the 35th International Conference on Computer Aided Verification*. Springer, 3–17. doi:10.1007/978-3-031-37703-7_1

[58] Rickard Nilsson. 2014. *ScalaCheck: The Definitive Guide*. Artima Press.

[59] A. Jefferson Offutt. 1992. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology* 1, 1 (1992), 5–20. doi:10.1145/125489.125473

[60] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology* 5, 2 (1996), 99–118. doi:10.1145/227607.227610

[61] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE, 75–84. doi:10.1109/icse.2007.37

[62] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* 112 (2019), 275–378. doi:10.1016/BS.ADCOM.2018.03.015

[63] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are Mutation Scores Correlated with Real Fault Detection?: A Large Scale Empirical Study on the Relationship Between Mutants and Real Faults. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 537–548. doi:10.1145/3180155.3180183

[64] Corina S. Păsăreanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: Integrating Symbolic Execution with Model Checking for Java Bytecode Analysis. *Automated Software Engineering* 20, 3 (2013), 391–425. doi:10.1007/S10515-013-0122-2

[65] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179. doi:10.1002/SPE.2346

[66] Hila Peleg, Dan Rasin, and Eran Yahav. 2018. Generating Tests by Example. In *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 406–429. doi:10.1007/978-3-319-73721-8_19

[67] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. tsDetect: An Open Source Test Smells Detection Tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1650–1654. doi:10.1145/3368089.3417921

[68] Mauro Pezzè and Michal Young. 2007. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons.

[69] Severin Primbs, Benedikt Fein, and Gordon Fraser. 2025. AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model. In *Proceedings of the 6th IEEE/ACM International Conference on Automation of Software Test*. IEEE, 12–23. doi:10.1109/ast66626.2025.00008

[70] Talia Ringer, Dan Grossman, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2017. A Solver-Aided Language for Test Input Generation. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 91:1–91:24. doi:10.1145/3133915

[71] Sreedevi Sampath, Renée C. Bryce, Gokulanand Viswanath, Vani Kandimalla, and Akif Günes Koru. 2008. Prioritizing User-Session-Based Test Cases for Web Applications Testing. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*. IEEE Computer Society, 141–150. doi:10.1109/ICST.2008.42

[72] Phillip Schanely. 2025. *CrossHair: An Analysis Tool for Python That Blurs the Line Between Testing and Type Systems*. https://github.com/pschanely/CrossHair Accessed: 2025-12-10.

[73] Ziqi Shuai, Zhenbang Chen, Yufeng Zhang, Jun Sun, and Ji Wang. 2021. Type and Interval Aware Array Constraint Solving for Symbolic Execution. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 361–373. doi:10.1145/3460319.3464826

[74] Sourcegraph. [n. d.]. *Sourcegraph Code Search*. https://sourcegraph.com/search Accessed: 2025-12-10.

[75] Chuyue Sun, Viraj Agashe, Saikat Chakraborty, Jubi Taneja, Clark W. Barrett, David L. Dill, Xiaokang Qiu, and Shuvendu K. Lahiri. 2025. ClassInvGen: Class Invariant Synthesis Using Large Language Models. In *Proceedings of the 2nd International Symposium on AI Verification*. Springer, 64–96. doi:10.1007/978-3-031-99991-8_4

[76] Yue Sun, Guowei Yang, Shichao Lv, Zhi Li, and Limin Sun. 2024. Concrete Constraint Guided Symbolic Execution. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. ACM, 122:1–122:12. doi:10.1145/3597503.3639078

[77] Suresh Thummalapenta, Madhuri R. Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. Retrofitting Unit Tests for Parameterized Unit Testing. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering*. Springer, 294–309. doi:10.1007/978-3-642-19811-3_21

[78] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*. Springer, 134–153. doi:10.1007/978-3-540-79124-9_10

[79] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized Unit Tests. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 253–262. doi:10.1145/1081706.1081749

[80] Deepika Tiwari, Yogya Gamage, Martin Monperrus, and Benoit Baudry. 2024. PROZE: Generating Parameterized Unit Tests Informed by Runtime Data. In *Proceedings of the 2024 IEEE International Conference on Source Code Analysis and Manipulation*. IEEE, 166–176. doi:10.1109/scam63643.2024.00025

[81] Ekaterina Tochilina, Vyacheslav Tamarin, Dmitry Mordvinov, Valentyn Sobol, Sergey Pospelov, Alexey Menshutin, Yury Kamenev, and Dmitry Ivanov. 2024. UTBot Python at the SBFT Tool Competition 2024. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*. ACM, 41–42. doi:10.1145/3643659.3643934

[82] Mark Trudinger and Henry Coles. [n. d.]. *Arcmutate: Mutation Testing Tools Improved*. https://www.arcmutate.com/ Accessed: 2025-12-10.

[83] Keita Tsukamoto, Yuta Maezawa, and Shinichi Honiden. 2018. AutoPUT: An Automated Technique for Retrofitting Closed Unit Tests Into Parameterized Unit Tests. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM, 1944–1951. doi:10.1145/3167132.3167340

[84] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. Refactoring Test Code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*. 92–95.

[85] Jifeng Xuan, Benoit Cornu, Matias Martinez, Benoit Baudry, Lionel Seinturier, and Martin Monperrus. 2016. B-Refactoring: Automatic Test Code Refactoring to Improve Dynamic Analysis. *Information and Software Technology* 76 (2016), 65–80. doi:10.1016/j.infsof.2016.04.016

[86] Xu Yang, Guofeng Zhang, Ziqi Shuai, Zhenbang Chen, and Ji Wang. 2025. Symbolic Execution of Floating-Point Programs: How Far Are We? *Journal of Systems and Software* (2025). doi:10.1016/J.JSS.2024.112242

[87] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *Comput. Surveys* 29, 4 (1997), 366–427. doi:10.1145/267580.267590

Received n/a; revised n/a; accepted n/a