Practical Disposable Regression Test Generation: An Experiment and Reflections

Anonymous Authors
Anonymous Department
Anonymous Institution
anonymous@anonymous.com

***Abstract*—Regression testing—the practice of re-executing test cases to reveal unintended changes in the behavior of a unit—is an expensive, human-intensive process. However, such effort *may not be necessary*. Each time that code is checked in, tests could be generated using existing code, executed against the newly committed version, and discarded after use. If effective, *disposable generation* could eliminate maintenance effort. However, questions remain about the efficacy of disposable testing—particularly given the strict time constraints necessary to fit within the continuous integration process.**

We have conducted an exploratory study to investigate these questions, and to determine challenges that must be overcome to perform practical disposable regression test generation. Using the EvoSuite generation framework, we have assessed the ability of suites to detect changes in 14 Java projects, examined the effect of targeting additional classes coupled to the changed classes, and investigated the human effort needed to assess test results. We have found that disposable test generation can be effective—36.73% of the regressions were detected, the inclusion of coupled classes can improve efficacy, and disposable testing often requires less inspection effort than traditional regression testing. Still, a number of major challenges must be addressed before it can truly replace human-guided regression testing.

***Index Terms*—Regression Testing, Disposable Testing, Search-based Test Generation, Automated Test Generation**

I. Introduction

As software evolves, it is important to ensure that the changes introduced to the source code do not *break functionality that has already been verified to work as intended*. This is a practice known as *regression testing*—the re-execution of test cases performed with the intention of revealing unintended changes in the behavior of a unit [25]. If the test suite passes before code changes, and fails after code changes, then it exposes a difference in program behavior. If that change was not intended, then the code has *regressed*.

Regression testing is a common activity in industrial software development [37], [28], and is commonly performed as part of continuous integration (CI). Each time that changes are checked into a source code repository, an automated daemon builds the project and executes a *regression suite* against the compiled code [10]. If potential regressions are revealed, then the production environment is rolled back to a working state, and the developers are informed of the regressions.

Regression testing is an expensive, human-intensive process [34]. For test cases to reveal regressions, they must be updated to reflect *intended* program changes or discarded as irrelevant. As the program evolves, and the size of the regression suite grows, this task becomes more arduous. Further, although test execution is automated by the CI daemon, the size of the regression suite can grow beyond the point of practicality—delaying the build process. As CI may be performed many times per day, significant delays in checking and deploying a build are unacceptable.

A number of techniques have been proposed to control this effort, many of which prioritize or select a subset of the regression suite for execution based on code or change coverage [39]. While such techniques can reduce the execution time needed for regression testing—and allow the human maintainer to make decisions—the fundamental cost of suite development and maintenance remains high.

However, such maintenance *may not be necessary*. Rather than generating and executing tests on the same version of a program, much of the research in automated test case generation posits a situation where tests are generated on a *fixed* program and executed against a *faulty version of the same program* (for example: [35], [16], [18]). This is done for a practical reason, as it eliminates the need to create a bespoke test oracle for each generated test case. If the test passes on the version it was generated on, and fails on the faulty version, then there is a high probability that the failure indicates that the fault was detected. In practice, this *is* regression testing—test efficacy is judged on the generated suite's ability to discern the differences between two program versions. This means that essentially any approach to automated generation can produce regression tests. That said, if automated generation is used to simply replace human-guided test creation, the fundamental maintenance challenge remains. In fact, it is magnified, given the readability challenges associated with generated test cases [2], [14]. However, another question can now be raised—*why maintain these tests at all?*

Each time that code is checked in, the CI daemon could generate tests using the latest working iteration of the code, execute these tests against the newly committed version, and report any detected regressions. After this process, the generated tests could simply be discarded. *If* tests can be generated and executed within the timing constraints imposed by the CI process, and *if* the same regressions can be reliably detected by the generated suites, then there is no need for manual regression test creation or maintenance. The human effort required for regression testing would be reduced to inspection of the failing test cases.

Shamshiri et al. recently coined a term for this idea—*disposable testing* [34]. The concept of disposable regression testing is promising, but has not yet been empirically explored, and questions remain about its potential efficacy. Can disposable testing be conducted effectively given the *practical* time constraints of the CI process? Will multiple rounds of generation reliably detect the same regressions? Which classes should be targeted for test generation—just those that changed, or classes that depend on the changed ones? Finally, does disposable testing *actually* reduce human effort, or does it shift where that effort is spent?

We have conducted an exploratory study to investigate these questions, and to assess the challenges that must be overcome to perform practical disposable regression test generation. Using the search-based EvoSuite test generation framework [14], we have generated suites under a realistic time budget and assessed their ability to cover code and detect changes between versions of 14 Java projects. We have also performed generation for optimized sets of classes including both the changed classes and additional coupled classes in order to assess the effect of including classes beyond those that have been directly changed. Finally, we have also assessed the human effort required to inspect failing test cases, as compared to traditional regression testing. Our goal is not to demonstrate that EvoSuite is the best possible means of disposable test generation. Rather, EvoSuite's maturity, reliability, and applicability [12], [13] give us baseline results that can be used to explore the questions highlighted above. Our study has found that:

- While there is room for improvement, disposable tests can detect sophisticated regressions. Overall, there is an average 22.69% likelihood of detection, and 36.73% of regressions were detected at least once.
  - For the regressions it is able to detect, disposable testing is relatively reliable, with an average likelihood of detection of 60.69%
- Generated suites achieve an average 60.14% Branch Coverage over the targeted classes. Only 1.86% coverage is lost on average over the changed version of each class.
- In many situations, the inclusion of coupled classes has no impact. Across all cases, there is an overall improvement of 3.79%.
  - However, when the additional classes have *any* impact, there is an average improvement of 120.26% and seven additional faults were detected. The inclusion of coupled classes could yield significant efficacy improvements if we can identify situations in advance where they would be useful, and improve coverage of the dependent lines of code.
- The potential cost savings are significant. Overall, the *upper* limit of human effort required for disposable testing is 25% less than the *lower* limit of human effort required for traditional regression testing.

Reflecting on our observations, we have identified five key challenges that, if overcome, may allow disposable testing to become a practical reality:

- Generated tests must be able to discern behavioral differences between versions of a class, despite strict generation and execution time limitations.
- Generated tests must detect regressions *reliably*. Each time a class regresses, that regression should be detected.
- To decide when to target additional classes, we must understand when their inclusion will be helpful.
- The efficacy of generation—when coupled classes are included as targets—may be improved if coverage is ensured of references to changed classes.
- Regardless of effort savings, disposable generation must yield *informative results* for the human inspector.

Although there is clear room for improvement, we believe these results are promising. Despite the strict constraints imposed by the CI process, our current automated test generation frameworks can detect sophisticated differences between versions of target classes—and can often do so with a reasonable degree of reliability. Our study offers baseline results that we hope will inspire new research advances, and shines a light on the challenges that must be overcome before disposable testing can replace human-guided regression testing.

II. Background

A. Unit and Regression Testing

Software tests can be written for many levels of granularity—from testing a single method to the interface of the complete system [29]. Arguably the most common form of test case is the *unit test* [33]. Unit tests are intended to test the functionality of one *unit* of code—typically a class—in isolation from the rest of the system [29]. Unit tests commonly consist of a series of method calls (the *test sequence*) and a *test oracle*, which issues a verdict on the correctness of the unit's behavior [5]. Unit testing frameworks exist for almost all programming languages, such as JUnit for Java [6], and are integrated into most development environments. One major reason for the prevalence of unit testing is that test execution can easily be automated. As discussed earlier, a common practice is to execute test suites during continuous integration.

Regression testing—the practice of re-executing test cases to verify that unchanged functionality continues to work as intended [25]—can be thought of as a form of unit testing. In fact, unit tests are commonly retained and used as regression tests [28]. Rather than a bespoke oracle, captured output of the latest working version of the class can serve as the oracle for regression tests. Any unexpected differences in output are likely to be regressions.

B. Search-Based Unit Test Generation

There are numerous forms of automated test generation. In this study, we make use of search-based generation. Test case creation can naturally be seen as a search problem [19]. Of the thousands of test cases that could be generated for any class, we want to select—systematically and at a reasonable cost—those that meet our goals [27], [1]. Given a well-defined testing goal, and a scoring function denoting

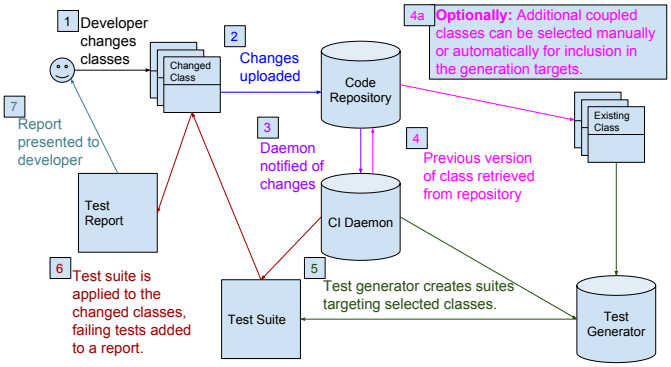

Fig. 1. An outline of disposable testing as part of a CI process.

closeness to the attainment of that goal—called a *fitness function*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*) [7]. Metaheuristics are often inspired by natural phenomena, such as swarm behavior [9] or evolution [20].

While the particular details vary between algorithms, the general process employed is as follows: (1) One or more test suites are generated, (2), the suites are scored using the fitness function, and (3), this score is used to reformulate the population for the next round of evolution. This process continues over multiple generations, ultimately returning the best-scoring suites. Search-based generation is usually used to optimize coverage of adequacy criteria—often based on code coverage [27]—as such criteria can be straightforwardly transformed into fitness functions [4].

Due to the non-linear nature of software, resulting from branching control structures, the input space of a real-world program is large and complex [1]. Metaheuristic search—by strategically sampling from that space—can scale to larger problems than many other generation algorithms [26]. Such approaches have been applied to a wide variety of testing goals and scenarios [1], including regression testing [40].

III. Study

Regression test creation, maintenance, and execution is an expensive, often human-intensive effort [37], [28], [34]. However, such effort may not be necessary. The majority of test generation research—implicitly or explicitly—posits a scenario that mirrors regression testing where tests are generated using a working version of the code and executed against a faulty—or, at least, different—version of the code.

It follows then, that human-guided regression testing could be replaced by disposable regression testing [34]. When a new version of a class is available, regression tests could be generated using the latest working iteration of the code, the suite could be executed against the newly committed version, and any discrepancies could be reported. Following this process, tests would be disposed of. Each time the class changes, this process is performed again—eliminating the need for maintenance. As tests are discarded, the total suite size can also be kept within a range where minimization and prioritization are no longer needed to control execution time.

Specifically, we define disposable testing as a form of automated regression testing taking place as part of the continuous integration (CI) process, as depicted in Figure 1. A developer changes classes and uploads their changes to the code repository. The CI daemon—a program controlling the overall CI process—is notified of changes. The daemon retrieves the latest working version of each changed class from the repository. Optionally, additional classes that depend on the changed classes can be selected either based on human selection or an automated optimization (see Section III-C). The daemon then activates the test generator, producing test suites that target each selected class. Those test suites are then applied to the new version of each class. Any test cases that fail are collected and presented to the developer as potential regressions. Note that—in our study—we assume that all behavior differences are unintended and due to regression. In practice, some filtering mechanism would be needed to differentiate intended differences from unintended differences.

The practice of disposable testing has not yet been empirically evaluated, and questions remain about its potential efficacy under the strict time constraints of the CI process. In particular, we wish to address the following research questions:

1) Given a practical time budget, does disposable test generation effectively detect regressions in the code?
2) When multiple trials are performed, how *reliably* does disposable generation detect the same regressions?
3) Given a practical time budget, do the generation suites effectively achieve coverage of the source code?
4) What effect does targeting additional coupled classes have on the efficacy of disposable testing?
5) What effect does disposable testing have on the human effort required to perform regression testing?

We have conducted an exploratory study to investigate these questions, and to assess the challenges that must be overcome to perform practical disposable regression test generation. Using the search-based EvoSuite test generation framework [14], we have performed generation under a time budget that would not delay continuous integration—in our study, two minutes per class—and assessed the ability of generated suites to detect changes between versions of 14 Java projects. To reiterate, our goal is not to demonstrate that EvoSuite is the best possible means of disposable test generation. Rather, we believe that EvoSuite—as a relatively mature, reliable, and effective tool [12], [13]—allows us to attain a reasonable baseline that can be used to investigate the questions highlighted above.

The first three questions allow us to explore the basic efficacy of disposable test generation. Under the conditions imposed by the CI process, does disposable generation—as performed by EvoSuite—detect potential regressions? More importantly, does it detect regressions *reliably*—if generation is performed multiple times, are regressions detected each time? While coverage does not ensure detection, suites that do not cover the code are unlikely to detect regressions. Therefore, we also assess Branch Coverage over both the original and modified version of each class.

An additional topic that we wish to explore is *which* classes should be targeted for test generation. Traditionally, tests are generated solely for the classes we know to contain faults. However, a variety of classes may depend on the changed classes, and by targeting these *coupled* classes, we may be more likely to detect regressions. Finally, we have also examined the human effort required to inspect failing test cases. Our hypothesis is that disposable testing sharply reduces human effort. However, if enough tests require inspection, we have simply shifted the burden. We compare the number of regressions reported by disposable testing—the upper bound on the human effort of disposable testing—to the number reported by the traditional regression suite—the lower bound on human-driven regression testing.

We have performed the following experiment:

1) **Collected Case Examples:** We have used 588 real faults, from 14 Java projects, to represent potential regression scenarios (Section III-A).
2) **Generated Test Cases:** For each fault, we generated 10 suites using the working version of each changed class. We allow a two-minute generation budget per targeted class (Section III-B).
3) **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are automatically removed (Section III-B).
4) **Assessed Detection Efficacy:** For each regression, we measure the proportion of test suites that detect the regression to the number generated (Section III-D).
5) **Recorded Branch Coverage:** For each regression, we measure the Branch Coverage over both versions of each class (Section III-D).
6) **Repeat Steps 2-4 For Groupings of Coupled Classes:** For each case example, we generate 10 optimized groupings of coupled classes. We then generate 10 suites targeting each grouping—in addition to the changed classes—measuring how the additional classes affect suite efficacy (Section III-C).

A. Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [23][1]. Currently, it consists of 597 faults from 15 projects: Chart (26 faults), Closure (133), CommonsCLI (24), CommonsCSV (12), CommonsCodec (22), CommonsJXPath (14), Guava (9), JacksonCore (13), JacksonDatabind (39), JacksonXML (5), Jsoup (64), Lang (65), Math (106), Mockito (38), and Time (27). For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the faults, and a list of classes and lines of code modified by the patch that fixes the fault. The Guava faults were omitted from this study, as its code uses Java 8 features not supported by the utility we used to produce groupings of coupled classes.

We have used the remaining 588 faults as example "regressions" for this study. While these examples do not necessarily represent actual regressions, we believe that these faults still serve as reasonable case examples for multiple reasons. First, each fault is associated with a pair of code versions that differ only by the minimum changes required to address the fault. These program versions were gathered from the actual version control system of each project. This means that, for each fault, we have two project versions that can be clearly compared, where behavioral differences are due to a small set of known changes. Additionally, we have access to the actual test suite of each project, including human-built tests that expose each fault. These "trigger tests" are, in most cases, tests that were actually added for the explicit purpose of regression testing. As a result, we can compare generated test suites to the actual regression suite for each case example.

[1]Available from http://defects4j.org

B. Test Suite Generation

The EvoSuite framework uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to the Defects4J dataset [35], [16]. In this study, we used EvoSuite version 1.0.5.

Tests are generated targeting Branch Coverage. Branch Coverage is satisfied if all control-flow branches are taken by at least one test—the suite contains at least one test whose execution evaluates the branch predicate to `true`, and at least one that evaluates the predicate to `false`. To guide the search, the fitness function calculates the *branch distance* from the point where the execution path diverged from the targeted branch. If an undesired branch is taken, the function describes how "close" the targeted predicate is to being satisfied.

Test suites are generated that target two groups of classes: (1) the classes that differ between the original and updated versions, and (2), the classes from Group 1 plus additional classes that are coupled to the changed classes (see Section III-C). In all cases, tests are generated using the version of each class known to be correct and applied to the changed version.

If disposable test generation is to be effective, it must be able to generate and execute test suites within a reasonable time budget—ideally, within the span of minutes. This is to prevent disruption to the CI process, where new versions of the system are built and deployed on a frequent basis. A "reasonable" budget is somewhat subjective, but generation budgets from 1-10 minutes per class are common in test generation research [35]. We have elected to impose a two minute budget per class, as such a budget has yielded reasonable results [16], [17], and should allow disposable testing to take place without unnecessarily delaying the CI process. To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each case example and grouping.

Generation tools may generate flaky (unstable) tests [35]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, all non-compiling test suites are

removed. Then, each remaining test suite is executed on the fixed version five times. If the test results are inconsistent, the test case is removed. This process is repeated until all tests pass five times in a row. On average, less than one percent of the tests are removed from each suite [16].

C. Optimizing Groups of Coupled Classes

When generating test suites, a set of classes must be chosen as targets. Often, generation is performed solely on the classes known to be faulty in order to control experiment cost. However, this does not mean that generation should be *restricted* to such classes. In most complex systems, classes depend on other classes to perform certain tasks. Such classes are said to be *coupled*, linked either unidirectionally or bidirectionally in their dependence on each other [30].

Coupling is especially relevant in the case of regression testing. An unchanged class that depends on a changed class may still exhibit unexpected behavior. By generating tests for classes that are coupled to the updated classes, we may be able to detect regressions that would otherwise be missed. Therefore, in addition to the standard generation scenario—targeting each changed class—we have also generated for groupings that include additional coupled classes.

In any complex system, more classes exist than could be realistically targeted for test generation. For example, in our case study, Closure has over 1,000 classes. At two minutes per class, it would take over 33 hours to generate tests—far past the practical limitations of regression testing, CI process or not. Therefore, in order to select optimized groupings of coupled classes, we have developed a simple framework that maps class dependencies and applies a genetic algorithm to form small, dense groups of coupled classes[2]. This framework:

1) Maps dependencies between classes. In this case, we consider dependencies to be either method calls or variable references from another system class.
2) A directed graph is created, where each class is a node, and each edge indicates a dependency. Any classes that have no dependencies and are not the target of a dependency will be filtered out from consideration at this stage. If no classes are coupled to a changed class, that class will still be added to the target list.
3) Groupings are formed by randomly selecting classes (up to a random set size). In this case, we generate a population of 1,000 groupings.
4) Each grouping is scored using the fitness function described below, and a new population is formed through retention of best solutions (10%), mutation (20%), crossover (20%), and further random generation (50%).
5) Evolution continues for until the time budget is exhausted—in this case, five minutes.
6) The best grouping is returned. The changed classes are added to that grouping.

[2]Available from https://github.com/XXXXXXX

	Number of Classes
Chart	18.63
Closure	73.92
CommonsCLI	1.91
CommonsCodec	3.51
CommonsCSV	3.40
CommonsJXPath	20.80
JacksonCore	5.03
JacksonDatabind	34.96
JacksonXML	2.80
Jsoup	26.34
Lang	7.12
Math	12.64
Mockito	34.43
Time	52.87

TABLE I
AVERAGE NUMBER OF CLASSES IN OPTIMIZED TARGET SETS.

The fitness function used to score groupings is:

$$F_G = \sqrt{\overline{size}^2 + \left(\overline{coverage} - 1\right)^2 + \overline{avg(distance)}^2} \quad (1)$$

That is, we prioritize groupings that are closer to a *sweet spot* of fewer classes ($size$), where the chosen classes are coupled to a large number of other classes ($coverage$), and where more classes are either coupled directly to the changed classes or through a small number of indirect dependency links (average $distance$). This should result in a relatively small grouping of classes that are densely coupled to each other and other classes. $\overline{x}$ is a normalized value $0 \leq \frac{x - min(x)}{max(x) - min(x)} \leq 1$. Scores range from $0 \leq F_G \leq \sqrt{3}$ and lower scores are better.

As this is a stochastic process, we generate 10 groupings per regression. In Table I, we list the average optimized grouping size for each system. In many cases, these groupings are larger than would be practical for the CI process. However, we still used them as they are useful for understanding whether there is *any* benefit to including coupled classes. We found that, in situations where coupling affects the results, only a small number of classes are useful, and they are closely linked to the changed classes. Therefore, these groupings could be easily pruned down to a size more appropriate for the CI process.

D. Data Collection

The effectiveness, for each fault, is judged in two ways. First, we measure whether each regression was detected at all. Second, we also calculate the likelihood of detection—the proportion of suites that successfully detect the regression to the total number of suites generated. This allows us to assess how reliably each regression is detected.

In addition, we have also measured the Branch Coverage attained by each suite on both the original and changed version of each targeted class. We have used EvoSuite's coverage measurement utility to measure coverage. This allows us to understand how well the targeted version of the class is covered and how well those tests hold up against the changes made to the class.

IV. RESULTS & DISCUSSION

We address our research questions as follows. In Section IV-A, we examine the efficacy and reliability of disposable testing (questions 1-3). In Section IV-B, we examine the effect

	Detection Likelihood	Number Detected
Chart	40.00%	16 (61.54%)
Closure	4.10%	15 (11.28%)
CommonsCLI	30.42%	10 (41.67%)
CommonsCodec	31.36%	12 (54.55%)
CommonsCSV	55.00%	10 (83.33%)
CommonsJXPath	21.43%	4 (28.57%)
JacksonCore	52.31%	8 (61.54%)
JacksonDatabind	13.60%	9 (23.08%)
JacksonXML	60.00%	3 (60.00%)
Jsoup	19.80%	23 (35.94%)
Lang	35.20%	32 (49.23%)
Math	28.68%	54 (50.94%)
Mockito	8.70%	4 (10.52%)
Time	34.40%	16 (59.26%)
Overall	22.69%	216 (36.73%)

TABLE II
AVERAGE LIKELIHOOD OF REGRESSION DETECTION AND THE NUMBER OF FAULTS DETECTED (ALONG WITH % DETECTED).

	Adjusted Detection Likelihood
Chart	57.78%
Closure	36.67%
CommonsCLI	73.00%
CommonsCodec	57.50%
CommonsCSV	66.00%
CommonsJXPath	75.00%
JacksonCore	71.25%
JacksonDatabind	58.89%
JacksonXML	100.00%
Jsoup	55.22%
Lang	71.56%
Math	56.30%
Mockito	82.50%
Time	58.13%
Overall	60.69%

TABLE III
AVERAGE LIKELIHOOD OF REGRESSION DETECTION IN SITUATIONS WHERE AT LEAST ONE SUITE DETECTED THE REGRESSION.

of adding additional coupled classes (question 4). Finally, in Section IV-C, we examine human effort (question 5).

A. Detection Efficacy, Reliability, and Coverage

In order to understand the applicability of disposable testing and the roadblocks to surmount, we require a baseline understanding of how effective the practice is at detecting regressions given the time constraints imposed by the CI process and the limitations of current test generation technology. In Table II, we list the number of regressions detected for each system, as well as the average likelihood of detection—the proportion of suites that detect each regression.

Overall, 216 of the 588 examples were detected, or 36.73%. On a per-system basis, the percent detected ranged from 10.52% (Mockito) to 83.33% (CommonsCSV). The average overall likelihood of detection is 22.69%—per system, ranging from 4.10% (Closure) to 60.00% (JacksonXML). There is clearly room for improvement, but these results are encouraging. The examples from Defects4J correspond to complex program changes, which may require intricate input and method call sequences to detect. These results are in-line with other studies on Defects4J [35], [16], [17], [15], [3]. To summarize:

> While there is room for improvement, disposable tests can detect sophisticated regressions. Overall, there is an average 22.69% likelihood of detection, and 36.73% of regressions were detected at least once.

	Branch Coverage (Fixed Version)	Branch Coverage (Regressed Version)
Chart	66.97%	63.98%
Closure	21.84%	22.26%
CommonsCLI	87.62%	85.76%
CommonsCodec	96.96%	95.17%
CommonsCSV	87.08%	86.58%
CommonsJXPath	67.54%	61.27%
JacksonCore	51.86%	50.47%
JacksonDatabind	52.33%	50.75%
JacksonXML	28.96%	26.52%
Jsoup	74.09%	68.71%
Lang	78.73%	78.68%
Math	73.87%	69.57%
Mockito	48.46%	-
Time	78.46%	77.04%
Overall	60.14%	59.02%

TABLE IV
AVERAGE BRANCH COVERAGE OVER THE FIXED AND REGRESSED VERSIONS OF EACH CLASS.

To gain a clearer picture of efficacy, we need to look at two additional factors—how reliably regressions are detected, and how effective the generated tests are at covering the code.

The likelihood of detection gives us an idea of the reliability. For any given regression, an average of 22.69% of the generated suites will detect it. However, this average takes into account situations where *regressions are never detected*. It is also useful to filter for regressions detected at least once. In Table III, we list the average likelihood of detection in such situations. Following this filtering, we can see that:

> If a regression can be detected, disposable testing will detect its reoccurrence with relatively reliable—with an average likelihood of detection of 60.69%.

On a per-system basis, the adjusted likelihood ranges from 36.67% (Closure) to 100.00% (JacksonXML).

In Table IV, we list the average Branch Coverage attained by suites over the fixed version (the generation target) and the regressed version of each class. Note that an issue with the build system for Mockito prevented coverage measurement on the regressed version for that system. We observe that:

> Generated tests attain an average of 60.14% coverage over the targeted classes. While some coverage is lost due to changes made to the regressed version, that loss is only 1.86% on average.

Per-system, average coverage over the targeted version ranges from 21.84% (Closure) to 96.96% (CommonsCodec). Improvements are again needed, but these figures are—in most cases—reasonable given the limited search budget.

On the whole, these results are encouraging. However, from them, we can identify two important, closely-related challenges to focus on. First:

Challenge: Generated tests must be able to discern behavioral differences between versions of a class, despite strict generation and execution time limitations.

Most test generation research implicitly follows a regression testing scenario, so almost any form of generation could be used to perform disposable testing. While we have used EvoSuite, a generic generation framework, test generation techniques have been proposed specifically for regression testing (as we will discuss further in Section V). Such techniques generally analyze both versions of the program, and as a result, they tend to require a high overhead on generation time. In many cases, the restrictions on the amount of time available during the CI process makes a heavyweight analysis impractical. We need to develop lightweight techniques that increase the likelihood of regression detection. In addition:

Challenge: Generated tests must detect regressions *reliably*. Each time a regression reappears, disposable testing should detect it.

Both challenges must be addressed before disposable testing can gain traction. One suggestion that has been proposed for the first challenge—and that could also address the second—is to employ *seeding* [34]. Seeding is the process of providing a pool of test input for the generation process. Rather than using randomly-chosen values to create the initial test suites, the generation framework makes use of the seeded values.

Seeding has been employed for various purposes in search-based generation [31]. It could have particular relevance to regression testing, as regression testing is a process that is repeated many times over the lifetime of a project. While we dispose of test *suites*, we could retain *input* that previously attained high levels of coverage. Even though the targeted class may have changed since the last time tests were generated, seeding can be used to kickstart coverage during a fresh generation cycle.

Because regression testing is performed repeatedly, seeding could lead to a steady rise in the average coverage level over time. Even if coverage is low when regression testing begins, it may become reliably high after a few rounds of disposable generation. While coverage alone does not ensure that regressions are detected [21], [18], it is a prerequisite to detection. Suites that more thoroughly explore the code are theoretically more likely to notice changes to that code.

Seeding could also address the second challenge—improving the reliability of detection. If tests detect a regression once, and their inputs are retained, then the same regression could be detected again. If we ensure such input is used each time we seed, generated tests may be trained towards the detection of known issues. If such regressions reappear, then the seeded tests should be able to detect them each time.

While seeding is only one route towards addressing such challenges, it is one we plan to explore in future work.

	Detection Likelihood (Changed-Only)	**Detection Likelihood (With Coupled)**
Chart	40.00%	42.58%
Closure	4.10%	5.10%
CommonsCLI	30.42%	30.42%
CommonsCodec	31.36%	35.55%
CommonsCSV	55.00%	58.50%
CommonsJXPath	21.43%	21.43%
JacksonCore	52.31%	52.31%
JacksonDatabind	13.60%	13.60%
JacksonXML	60.00%	60.00%
Jsoup	19.80%	21.70%
Lang	35.20%	35.50%
Math	28.68%	29.29%
Mockito	8.70%	8.70%
Time	34.40%	35.90%
Overall	22.69%	23.55%

TABLE V
AVERAGE LIKELIHOOD OF DETECTION WHEN ONLY THE CHANGED CLASSES ARE TARGETED AND WHEN COUPLED CLASSES ARE INCLUDED.

	Detection Likelihood (Changed-Only)	**Detection Likelihood (With Coupled)**
Chart	15.00%	27.50%
Closure	30.00%	62.50%
CommonsCodec	13.33%	44.00%
CommonsCSV	30.00%	51.00%
Jsoup	15.00%	27.80%
Lang	10.00%	30.00%
Math	6.67%	28.33%
Time	25.00%	44.00%
Overall	18.26%	40.22%

TABLE VI
AVERAGE LIKELIHOOD OF DETECTION WHEN ONLY THE CHANGED CLASSES ARE TARGETED AND WHEN COUPLED CLASSES ARE INCLUDED—OMITTING CASES WHERE COUPLED CLASSES HAVE NO EFFECT.

B. The Effect of Adding Coupled Classes

In a complex object-oriented system, classes tend to depend on the functionality of other classes. Therefore, the behavior of the called class may be impacted by changes to other, *coupled*, classes. When generating tests, we may be better able to detect certain regressions by generating tests for not just the changed classes, but classes that—in turn—depend on the changed classes.

In Table V, we compare the average likelihood of detection between the normal case—where only the changed classes are targeted—and when we generate for a set of targets including coupled classes. From this table, we can see that there is often *some* improvement, but the overall effect is relatively minimal. The inclusion of coupled classes fails to improve results for six systems—CommonsCLI, CommonsJXPath, JacksonCore, JacksonDatabind, JacksonXml, and Mockito. For the others, we see average improvements of up to 13.36%. Overall, the average improvement from including coupled classes is only 3.79%. There is some benefit, but it is generally not worth the additional time investment.

For the majority of faults, the inclusion of additional classes has no impact. Therefore, it is also worth focusing on cases where they do have an impact. Table VI lists the average likelihood of detection for the 23 faults where the inclusion of coupled classes have an impact. These filtered results show that when additional classes have any impact, it is a major one. Overall, in such cases, the results improve by 120.26%. In fact, seven faults were *only* detected by including coupled classes. Therefore, we observe that:

In many situations, the inclusion of coupled classes has no impact (an overall improvement of 3.79%). However, when the additional classes have *any* impact, there is an average improvement of 120.26%. Seven additional faults were detected by including coupled classes.

While the addition of classes can be very powerful, it is also very expensive given that—by default—the same amount of time is devoted to generating test cases for each class. We hypothesize that two challenges must be addressed in order to better harness coupled classes.

Challenge: To decide when to add additional targets, we must understand when their inclusion will be helpful.

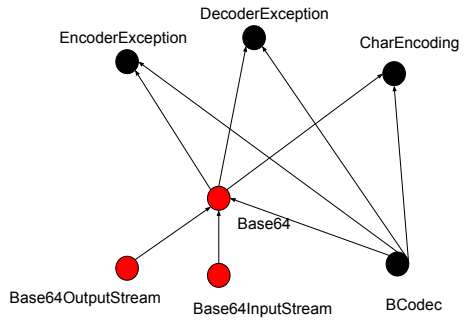

Fig. 2. Partial visualization of coupling in the CommonsCodec project. Target class—Base64—and relevant coupled classes are colored red.

Figure 2 depicts a selection of connected classes in the CommonsCodec project. Three examples in CommonsCodec—centering around changes to the `Base64` class (faults 12, 15, and 20[3])—see improved efficacy from the inclusion of coupled classes `Base64InputStream` and `Base64OutputStream`. Tests generated solely to target `Base64` are able to detect all three faults, but not reliably. The incorporation of these two coupled classes greatly increases the likelihood of detection. Another class—`BCodec`—is also coupled to `Base64`, but does not contribute to suite efficacy.

These three examples are interesting because the two additional classes are not just coupled through in-code dependencies, but all three are linked by a common conceptual purpose—encoding binary data by treating it numerically and translating it into a base 64 representation. It follows then that one option for incorporating coupled classes would be to periodically present human developers with coupling information and ask them to filter groupings. Each time that any class in that grouping is altered, those coupled classes could be included in generation.

[3]https://github.com/Greg4cr/defects4j/blob/master/framework/projects/CommonsCodec/patches/[12/15/20].src.patch

Of course, not all situations where coupling assists are as straightforward as the CommonsCodec example. For instance, consider fault 31 for the Math system[4]. In this case, the regression is in class `ContinuedFraction`. However, disposable testing never detect the issue when targeting that class. Instead, the fault is only detected when tests are generated for `Gamma` (coupled to `ContinuedFraction`) and `GammaDistribution` (coupled to `Gamma`). The reason the coupled classes are useful is likely because they provide guidance to the generator in how to make use of `ContinuedFraction`. Exposing the fault requires setting up a series of values and calling `ContinuedFraction.evaluate(...)` on those values. By attaining coverage of `Gamma`, EvoSuite is able to set up and execute the functionality of `ContinuedFraction`. Without that guidance, it struggles.

While only a small number of classes are coupled to `ContinuedFraction`, there is not a common conceptual connection like with the CommonsCodec example above. In retrospect, we can explain these situations. However, more research is needed to recognize patterns in when the inclusion of coupled classes is beneficial.

Further, asking developers to name useful couplings creates additional maintenance effort. Therefore, we also need further research into automated means to suggest and prune couplings. One thing that could be done is to automatically generate groupings as we have in this study, and—if these groupings prove to not be useful—steadily remove pairings. For instance, we saw no scenarios where coupled classes improved the likelihood of detection for the three Jackson systems. In the best cases, a few additional tests failed. For cases like this, we could prune away coupled classes once a threshold of "usefulness" is passed.

At the same time, we should endeavor to make the inclusion of coupled classes more useful by focusing on increased coverage of those dependencies:

Challenge: The efficacy of generation—when coupled classes are included as targets—may be improved if coverage is ensured of references to changed classes.

Currently, test generation for each class is an entirely independent process. While the attained branch coverage may be relatively high for each targeted class, we have no guarantee that dependencies *between classes* are covered. Steps could be taken to improve coverage of such dependencies by considering coverage of dependencies between classes as a special form of Line Coverage. This "Coupling Coverage" could be used to prioritize suites that attain a higher coverage of the specific lines of code in a class that are dependent on data or functionality from a changed class. When targeting a changed class, the test generator could focus on the normal generation target, i.e., Branch Coverage. However, when targeting a class

[4]https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/31.src.patch

	Number of Failing Tests	Num. Failing, with Coupled Classes	Number of Trigger Tests
Chart	2.23	2.64	4.44
Closure	1.17	1.25	6.33
CommonsCLI	1.23	1.27	1.60
CommonsCodec	1.35	1.56	2.36
CommonsCSV	1.25	1.24	1.80
CommonsJXPath	1.10	1.10	1.25
JacksonCore	2.02	2.06	1.40
JacksonDatabind	1.43	1.43	1.22
JacksonXML	1.67	1.67	1.33
Jsoup	2.19	2.46	1.65
Lang	2.05	2.09	1.53
Math	1.56	1.56	1.46
Mockito	1.00	1.01	2.50
Time	1.65	1.70	2.75
Overall	1.65	1.75	2.21

TABLE VII
AVERAGE NUMBER OF FAILING TESTS WHEN DISPOSABLE GENERATION IS PERFORMED (WITH AND WITHOUT COUPLED CLASSES) AND THE NUMBER OF HUMAN-WRITTEN TEST CASES THAT DETECT THE REGRESSION.

that depends on a changed class, this "Coupling Coverage" could be used instead of, or in addition to, Branch Coverage.

Recent work has found that combinations of orthogonal coverage criteria can be more effective than individual criteria [17]. For example, combining Branch and Exception Coverage yields test suites that both cover the code and force the program into unusual configurations—improving the likelihood of fault detection. "Coupling Coverage" could be thought of as another situationally-appropriate orthogonal criterion. Rather than generating tests using Branch Coverage alone, the generator could combine Branch and "Coupling Coverage" when targeting coupled classes—potentially creating suites that are especially effective at exploiting dependencies between classes, and in turn, at detecting regressions. In future work, we will explore the forms such a criterion could take.

C. The Human Effort Requirements of Disposable Testing

The human effort required to perform regression testing and maintain the regression suite can be tremendous [28], [34]. In theory, disposable testing can almost entirely remove that burden. Rather than having to maintain a regression suite, select a subset for execution, and inspect the results, the human effort is spent solely on inspection of the failing test cases.

However, if enough generated tests fail, this can still be a significant task. Further complicating matters is the fact that generated test cases are notoriously difficult for human testers to understand [2], [14]. If disposable generation regularly produces enough failing test cases, then the human effort may simply shift from maintenance to inspection.

In order to assess the human effort required to inspect the results of disposable testing, we have measured the average number of failing test cases each time a regression is detected. Cases where regressions were not detected by any of the generated suites are uninformative, and were ignored when calculating this average. The results are listed, for each system—both with and without coupled classes—in Table VII. In disposable testing, the number of cases that must be inspected represents the *upper bound* on human effort [34].

It is difficult to estimate the human effort required for traditional regression testing. However, we can use the same

```
@Test(timeout = 4000)
public void test081()  throws Throwable  {
    TokeniserState tokeniserState0 = TokeniserState.Doctype;
    CharacterReader characterReader0 =
        new CharacterReader("required");
    characterReader0.consumeTo("<C&[c8V`{Z]&o");
    ParseErrorList parseErrorList0 =
        new ParseErrorList(70, 70);
    Tokeniser tokeniser0 = new Tokeniser(characterReader0,
                                         parseErrorList0);
    tokeniserState0.read(tokeniser0, characterReader0);
    assertEquals(2, parseErrorList0.size());
}
```

Fig. 3. A regression-detecting test for the Jsoup system.

concept—the number of failing tests that require inspection—as a *lower bound* on the human effort required for regression testing [34]. In this traditional scenario, humans must also perform suite maintenance, which is the bulk of the required effort. Each example from the Defects4J dataset includes one or more *trigger tests*, human-written test cases that detect the issue. These tests are the ones that would fail in a regression scenario. Therefore, we can compare the average number of disposable tests that fail to the average number of trigger tests in order to assess the baseline human cost of disposable testing. The average number of trigger tests is listed in Table VII.

From Table VII, we can see that the inspection effort between disposable testing—as performed using EvoSuite—and manually-developed test cases is quite similar, in numeric terms. For eight of the fourteen systems, on average, *fewer* tests fail in the automated suite than would in the manually-developed suite. Overall, there is an average 25.34% decrease in effort for changed classes only, and a 20.81% decrease in effort when coupled classes are included. In the remaining cases, the additional inspection effort is generally minor—with a maximum increase of 32.73% in inspection effort with changed classes alone and 49.09% with coupled classes.

> Overall, the *upper* limit of human effort required for disposable testing—when generating for changed classes only—is 25% less than the *lower* limit of human effort required for traditional regression testing.

This is not a precise measurement, as the inspection effort required for an automatically-generated test case could be assumed to be higher than for a human-created test case [2], [14]. Therefore, while disposable testing clearly has great potential for lowering the cost of regression testing:

> **Challenge:** Regardless of human-effort savings, disposable generation must yield *informative results*.

A typical generated test can be see in Figure 3. This test detects the regression, but why this input and list of method calls trigger the issue may not be obvious. If disposable testing is to be successful, then the test input that fails must be readable and the test cases must follow a sensible series of

steps. If regressions occur, the failing test suite must help the human diagnose what is wrong with the program. Efforts must be made to both control the number of failing test cases, and ensure that the generated test cases are actually useful for human inspectors. Our results are encouraging in terms of the former case, but work is still required in regard to the latter.

V. Related Work

Regression testing has been a common practice in software development for decades [25], and naturally, a large body of research exists on aspects of the problem. Two subtopics in regression testing are particularly relevant to this work—automated regression test generation, and management of the burden of regression testing.

First, there are a variety of test generation techniques that have been proposed with the specific purpose of generating regression tests [24], [8], [36], [11]. All of these techniques are designed to expose differences between two versions of a system. For example, differential testing—the most similar approach to disposable generation—generates tests for both versions of the software, executes tests on the other, and checks for failures in both directions [11]. Chen et al. use an Extended Finite State Machine model and a dependence analysis [8]. The DiffGen framework adds new branches designed to expose differences between versions [36].

Our study is focused on the use of automated generation to remove the human burden, regardless of the source of the test cases. While we did not use an approach specifically aimed at regression testing, the above approaches could be used for disposable testing, and may even outperform a generic framework like EvoSuite—as long as such tools can perform within the time limits imposed by the CI process. However, as approaches specifically targeted towards regression testing tend to analyze both versions of each class, they generally require additional overhead. For instance, differential testing requires twice the generation and execution time, as it generates separate suites for both versions of each class. To give another example, model-based approaches introduce the additional human burden of model construction and upkeep. Therefore, while approaches targeted specifically towards regression testing may be more effective, they carry costs that must be considered when weighing their practical applicability.

Many approaches have been proposed to manage the size of the regression test suite, and thus, to reduce the human burden of suite maintenance [39]. Most techniques revolve around suite reduction through elimination of redundant cases [32], selection of subsets of test cases to execute [38], or prioritization of tests based on probability of fault detection [22]. If disposable testing is effective, tests need not be maintained at all. Still, if a human-built regression test suite is used, such techniques are useful in reducing the maintenance effort.

VI. Threats to Validity

External Validity: Our study has focused on a relatively small number of systems. Nevertheless, we believe that such systems are representative of—at minimum—other small to medium-sized Java systems. We believe that Defects4J offers enough examples to generalize our results to similar projects.

We have used a single test generation framework, EvoSuite. There are many methods of generation, which may yield different results. Our goal was to provide baseline data to examine the problem of disposable generation. We have chosen EvoSuite as it is a mature tool, capable of handling a wide variety of systems [12], [13]. EvoSuite has also yielded the best performance thus far on the studied systems [35]. We believe that EvoSuite is a reasonable choice given our goals.

To control experiment cost, we have performed ten trials for each example and class grouping. Similarly, we have only generated 10 groupings per example. It is possible that larger sample sizes in each area may yield different results. However, this process still required the generation of thousands of test suites (at two minutes per class), and there was significant overlap between groupings for the majority of examples. Therefore, we believe that our results are sufficiently stable.

VII. Conclusions

Regression testing—the practice of re-executing test cases to reveal unintended changes in the behavior of a unit—is an expensive, human-intensive process. However, such effort *may not be necessary*. Each time that code is checked in, tests could be generated using the existing code, executed against the newly committed version, and discarded after use. If effective, *disposable generation* could eliminate maintenance effort. However, the practice of disposable regression testing has not been empirically evaluated, and questions remain about its efficacy—particularly given the strict time constraints necessary to fit within the continuous integration process.

We have conducted an exploratory study to investigate these questions. We have found that disposable test generation can be effective—36.73% of the complex, real-world regressions were detected, the inclusion of coupled classes can improve efficacy, and disposable testing often requires less inspection effort than traditional regression testing. Although there is clear room for improvement, we believe these results are promising. Despite the strict constraints imposed by the CI process, automated test generation frameworks can detect sophisticated differences between versions of target classes—and can often do so with reasonable reliability. Still, challenges must be overcome before disposable testing can replace human-guided regression testing.

Our study offers baseline results that we hope will inspire new research advances. In future work, we will further explore how actions such as seeding can improve coverage and reliability over time, how we can ensure coverage of couplings between classes, and how we can improve the readability and usability of generated test suites.

VIII. Acknowledgements

This work is supported by National Science Foundation grant XXX-XXXXXXX.

REFERENCES

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, 2010.

[2] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)—Software Engineering in Practice Track (SEIP)*, ICSE 2017, New York, NY, USA, 2017. ACM.

[3] H. Almulla, A. Salahirad, and G. Gay. Using search-based test generation to discover real faults in Guava. In *Proceedings of the Symposium on Search-Based Software Engineering*, SSBSE 2017. Springer Verlag, 2017.

[4] A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147, 2013.

[5] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[6] K. Beck, E. Gamma, D. Saff, and M. Clark. JUnit unit testing framework. http://www.junit.org, 2015.

[7] L. Bianchi, M. Dorigo, L. Gambardella, and W. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8(2):239–287, 2009.

[8] Y. Chen, R. L. Probert, and H. Ural. Model-based regression test suite generation using dependence analysis. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, A-MOST '07, pages 54–62, New York, NY, USA, 2007. ACM.

[9] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[10] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[11] R. B. Evans and A. Savoia. Differential testing: A new approach to change detection. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, ESEC-FSE companion '07, pages 549–552, New York, NY, USA, 2007. ACM.

[12] G. Fraser and A. Arcuri. Evosuite: On the challenges of test case generation in the real world. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 362–369. IEEE, 2013.

[13] G. Fraser and A. Arcuri. Evosuite at the sbst 2017 tool competition. In *Proceedings of the 10th International Workshop on Search-Based Software Testing*, SBST'17, pages 39–42. IEEE, 2017.

[14] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, 24(4):23:1–23:49, Sept. 2015.

[15] G. Gay. Challenges in using search-based test generation to identify real faults in mockito. In *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, pages 231–237, Cham, 2016. Springer International Publishing.

[16] G. Gay. The fitness function for the job: Search-based generation of test suites that detect real faults. In *Proceedings of the International Conference on Software Testing*, ICST 2017. IEEE, 2017.

[17] G. Gay. Generating effective test suites by combining coverage criteria. In *Proceedings of the Symposium on Search-Based Software Engineering*, SSBSE 2017. Springer Verlag, 2017.

[18] G. Gay, M. Staats, M. Whalen, and M. Heimdahl. The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on*, PP(99), 2015.

[19] M. Harman and B. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.

[20] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[21] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

[22] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.

[23] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[24] B. Korel and A. M. Al-Yami. Automated regression test generation. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 143–152, New York, NY, USA, 1998. ACM.

[25] H. K. N. Leung and L. White. Insights into regression testing [software testing]. In *Proceedings. Conference on Software Maintenance - 1989*, pages 60–69, Oct 1989.

[26] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 436–439, Washington, DC, USA, 2011. IEEE Computer Society.

[27] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[28] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Commun. ACM*, 41(5):81–86, May 1998.

[29] M. Pezze and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.

[30] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 469–478, Sept 2006.

[31] J. M. Rojas, G. Fraser, and A. Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, 2016. stvr.1601.

[32] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[33] P. Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, July 2006.

[34] S. Shamshiri, J. Campos, G. Fraser, and P. McMinn. Disposable testing: Avoiding maintenance of generated unit tests by throwing them away. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 207–209, Piscataway, NJ, USA, 2017. IEEE Press.

[35] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE 2015, New York, NY, USA, 2015. ACM.

[36] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 407–410, Washington, DC, USA, 2008. IEEE Computer Society.

[37] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings The Eighth International Symposium on Software Reliability Engineering*, pages 264–274, Nov 1997.

[38] S. Yoo and M. Harman. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, 83(4):689 – 701, 2010.

[39] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[40] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *ACM International Conference on Software Testing and Analysis (ISSTA 09)*, pages 201–212, Chicago, Illinois, USA, 19th – 23rd July 2009.