Disposable Testing: Avoiding Maintenance of Generated Unit Tests by Throwing Them Away

Sina Shamshiri, José Campos, Gordon Fraser, and Phil McMinn
Department of Computer Science, University of Sheffield, UK
{sina.shamshiri, jose.campos, gordon.fraser, p.mcminn}@sheffield.ac.uk

***Abstract*—Developers write unit tests together with program code, and then maintain these tests as the program evolves. Since writing good tests can be difficult and tedious, unit tests can also be generated automatically. However, maintaining these tests (e.g., when APIs change, or, when tests represent outdated and changed behavior), is still a manual task. Because automatically generated tests may have no clear purpose other than covering code, maintaining them may be more difficult than maintaining manually written tests. Could this maintenance be avoided by simply generating new tests after each change, and disposing the old ones? We propose *disposable testing*: Tests are generated to reveal any behavioral differences caused by a code change, and are thrown away once the developer confirms whether these changes were intended or not. However, this idea raises several research challenges: First, are standard automated test generation techniques good enough to produce tests that may be relied upon to reveal changes as effectively as an incrementally built regression test suite? Second, does disposable testing reduce the overall effort, or would developers need to inspect more generated tests compared to just maintaining existing ones?**

```
String string0 = "Z,˜jsZ/7'{p!wd";
int int0 = 0;
SimpleTimeZone simpleTimeZone0 = new SimpleTimeZone(int0,
     string0);
Locale locale0 = Locale.GERMAN;
String string1 = "⋆z";
FastDatePrinter fastDatePrinter0 = new FastDatePrinter(
     string1, simpleTimeZone0, locale0);
MockGregorianCalendar mockGregorianCalendar0 = new
     MockGregorianCalendar(locale0);
String string2 = fastDatePrinter0.format((Calendar)
     mockGregorianCalendar0);
assertEquals("⋆GMT", string2);
```

Figure 1: A test case generated by EvoSuite that can detect a bug in Apache Commons Lang (Defects4J, Lang-8) [9]

I. Introduction

As software programs evolve over time, tests are used to check that existing functionality is not broken, and to capture the behavior of newly introduced functionality. In the context of object-oriented programming, these tests are implemented as automated unit tests that can be frequently and quickly executed. Every time the program is changed, the tests are re-executed. If a test fails after a change, then it exposes a difference in behavior. If the difference is intended, then the test needs to be updated to reflect the correct behavior, else the test has revealed a regression fault that needs to be fixed.

Because deriving a good set of unit tests is difficult, tests can be generated automatically instead. A standard approach to do so is to take a version of a class as input, use some technique to exercise a wide range of behavior (e.g., randomly [7], or driven by code coverage [3]), and then to add test assertions that capture the current behavior of the class under test. The resulting tests need to be maintained alongside the evolving program, just like manually written tests. However, maintaining generated tests can be tedious and challenging, since they are often lengthy and have no clear purpose. For example, Figure 1 shows a test case generated by EvoSuite [2] for the Apache Commons Lang library. This is an effective test, since it succeeds at revealing bug Lang-8 from the Defects4J [5] repository of bugs. However, the non-sensical string input and seemingly arbitrary combination of calls make it difficult to discern what the objective of the test is — a problem that is inherent in using any automatic unit test generation tool.

This leads us to the question of whether developers actually need to keep and maintain automatically generated tests. We propose *disposable testing* as an alternative approach: Instead of maintaining tests, completely new tests are generated every time the program under test is changed. Developers are only shown the tests that reveal a behavioral difference caused by the change. They then decide whether this difference is intended or not — as per usual following the execution of a regression test suite. Following this, the generated tests are thrown away.

In Figure 2 we illustrate the approach in a practical setting. After a new change has been made to the program (e.g., new code is committed to the version control system), a test generation tool generates tests intended to reveal behavioral differences between the previous and changed versions of the program. If a difference is found, a developer can inspect the tests to find if a regression has occurred. If so, the tests may be used to identify and fix the fault. The tests are then discarded.

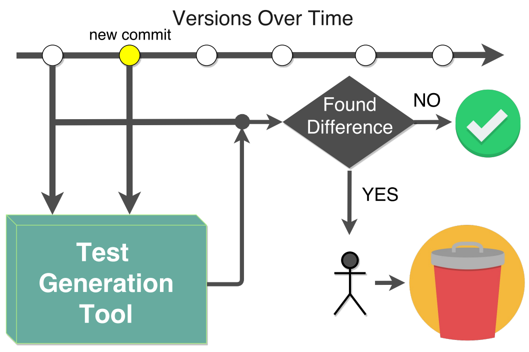

Figure 2: Overview of the process of disposable testing

The concept of disposable testing may at first seem counter-intuitive to developers, who usually like to keep as many tests as possible. Indeed, the idea of disposable testing raises several questions: Can generated tests find all the changes between two versions of a program? By throwing away all tests rather than adding to a test suite, do we run the risk of missing bugs? Would we have to inspect more test cases overall when applying disposable testing? That is, for disposable testing to be feasible and practical, two conditions need to hold:

1) We must be able to generate effective change-revealing tests on demand.
2) The manual effort involved must be less than that for a traditional "generate-and-maintain" approach.

The next section discusses how disposable testing may be applied in practice, and how we might evaluate its effectiveness.

II. Research Challenges

***Challenge 1**: Generating effective change-revealing tests*

The first challenge lies in generating effective tests on demand, that is, after a change has been made. To make the idea of disposable testing work, automated test generation techniques need to be as effective at revealing behavioral differences as a well-maintained regression test suite. A number of automated unit test generation tools and techniques exist. However, they tend to be directed towards generating high coverage test suites.

Although coverage-driven test generation approaches have been shown to be effective at finding real faults (e.g., [9]), coverage alone is not a strong indicator of the effectiveness of the generated tests [4]. However, traditional test generation approaches usually rely on only one version of a program, whereas in the regression testing scenario considered by disposable testing we always have *two* versions of a program — before and after a change. This raises the question of whether *Differential Testing* [1], [6] may be better suited to implement disposable testing. With differential testing, a test generator receives two program versions, before and after a change, as input, and derives tests that demonstrate behavioral differences.

Therefore, to address the first research challenge, we need to empirically evaluate (1) whether using differential testing can generate tests that are more effective at revealing behavioral differences than coverage based tests, and (2) whether behavioral differences can be found reliably enough to a level at which it could be considered that "good" tests can be generated at *any* time. The latter result is important since it means that tests could be regenerated whenever they are needed, and thereby disposed of following their inspection, rather than being kept and maintained as part of an evolving test suite.

However, test generation does not only need to be effective, but also efficient enough to provide quick feedback to developers. While generated tests are not maintained during disposable testing, throwing these tests away does not mean that the data and insights gained by the tools internally (e.g., symbolic insight on how to cover certain branches, or test data for seeding [8]) need to be discarded as well. By keeping this information internally, test generation tools can potentially become quicker and more effective.

***Challenge 2**: Is the maintenance effort really reduced?*

With disposable testing, generated tests are not integrated into the test suite, and so any maintenance effort related to these tests is avoided. However, effort is still required to inspect the change-revealing tests before they are disposed. As with traditional regression test suites, a developer needs to inspect test cases failures to determine whether it is because of an intended change or a regression fault. The question, therefore, is whether disposable testing will result in an increase in the number of tests that need to be inspected, compared to a traditional generate-and-maintain approach.

The manual effort required for a traditional generate-and-maintain approach involves both inspecting failing tests *as well as* maintaining the test code. The best way to directly compare maintenance effort for such an approach versus the effort spent on inspecting tests with disposable testing would be to perform a controlled human study. However, an approximate comparison could be performed by counting the number of tests that need to be *inspected* when applying the two different approaches. We can make a conservative comparison by assuming that every inspected test in a traditional generate-and-maintain approach reveals intended behavior, and the maintenance action consists of deleting the test. Although in practice tests may be modified and retained, this scenario gives us a lower bound on the maintenance effort. This is because if tests were retained rather than deleted, test suites would grow bigger over time, increasing maintenance effort.

For disposable testing, we can assume a scenario where developers have to inspect all behavioral differences. The number of failing tests in this case should provide an upper bound for the human effort of disposable testing. Comparing this against the lower-bound of the maintenance effort in the traditional generate-and-maintain approach provides us a conservative indication of the extent to which manual effort may be reduced through disposable testing.

III. Conclusions and Future Work

In this paper we proposed *disposable testing* as an alternative way of using automated test generation tools: Instead of generating unit tests automatically and integrating them into the code base, disposable testing involves generated new tests every time a program is changed. Tests that reveal changes between the two versions of the program (original and changed) may be inspected by developers and then thrown away. The advantage of disposable testing is that it avoids the effort of maintaining automatically generated test code, which tends to be difficult for humans to understand.

In order to implement disposable testing and to demonstrate its feasibility, there are a number of challenges that need to be overcome. We plan to investigate these challenges based on the approaches outlined in this paper. We will further investigate refinements of existing test generation approaches, in order to develop new, more effective testing techniques that will make disposable testing as efficient as possible.

References

[1] 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*, pages 549–552. ACM, 2007.

[2] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proc. of the Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419. ACM, 2011.

[3] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)*, 39(2):276–291, 2013.

[4] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. of the Int. Conference on Software Engineering (ICSE)*, pages 435–445. ACM, 2014.

[5] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440. ACM, 2014.

[6] W. M. McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1):100–107, 1998.

[7] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 815–816. ACM, 2007.

[8] J. M. Rojas, G. Fraser, and A. Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability (STVR)*, 2016.

[9] 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 *Proc. of the Int. Conference on Automated Software Engineering (ASE)*. IEEE, 2015.