Beyond coverage: what lurks in your test suites?

Patrick Lam

August 12, 2014

Abstract

Coverage is a standard metric used to evaluate test suites. But other properties matter too!

We are investigating 10 open-source test suites. How important is cloning (cut-and-paste programming) in test suites? How complex are test cases? Do they use a lot of mock objects? Do the comments make sense?

We all want "better" test suites. But what makes for a good test suite? Certainly, test suites ought to aim for good coverage, at least at the statement coverage level. To be useful, test suites should run quickly enough to provide timely feedback.

My talk will investigate a number of other dimensions on which to evaluate test suites. I claim that better test suites are more maintainable, more usable (for instance, because they run faster, or use fewer resources), and have fewer unjustified failures.

In my talk, I'll present and synthesize facts about 10 open-source test suites (from 8,000 to 246,000 lines of code) and evaluate how they are doing. This includes the standard measures of test suite goodness: size, coverage (statement, class hierarchy) and running time. But we'll also explore other aspects of these test suites: their static complexity; whether they are deterministic (or prone to nondeterministic failures!); their resource (memory, I/O) usage; potential for optimization; the number of test clones that we detected; and their refactorability.

Here's an overview of what we'll hear about:

- Size, coverage, running time: easy to measure using standard tools.
- Static complexity: Are tests just straight-line code? Do they contain loops and branches? What about method calls? How much virtual dispatch exists?
- Determinism: Flaky tests are terrible to deal with. I've done some past work on detecting nondeterminism and will attempt to apply it to test cases, if time permits.

- Resource usage: Running tests takes time (of course, faster is better). But it also uses memory and may make I/O calls (or use mock objects to avoid said calls). We'll look into how many resources our test suites consume while running, and explore avenues for reducing resource consumption.
- Optimization potential: In addition to reducing resource consumption, how parallelizable are these test suites? Can they effectively be run on clusters?
- Test clones: I'm carrying out ongoing research on test clone detection (see below). This talk will briefly outline results to date.
- Refactorability: The obvious next step after detecting clones is to refactor them. We'll talk about that too.

I've also included, below, the introduction from our planned ICSE submission about detecting test clones. It has a different focus from the proposed GTAC talk, but there are some ideas in common.

1 Introduction from ICSE submission

Modern software systems often use unit tests to ensure proper code behaviour. Ideally, a suite of unit tests works together to cover the system's functionality. However, achieving acceptable coverage using popular unit testing frameworks, such as JUnit, often requires the use of repetitive boilerplate code. The problem is that traditional unit test design recommends self-contained test classes—each test class must run independently and take no input parameters. The easiest way to create such self-contained test classes is to clone boilerplate test code, which increases test maintenance overhead and propagates any pre-existing errors.

Our goal is to facilitate the refactoring of suites of unit tests by detecting test clones. In this paper, we describe the design and implementation of a tool that identifies cloned unit tests and provides contextual information about these clones to the developer.

A number of technologies already exist for implementing test refactoring. Tillman and Schulte have proposed *parametrized unit tests (PUTs)* to increase the expressive power of unit testing and to enable test reuse [TS05]. Similarly, Saff proposed *theories* to simplify and increase the robustness of unit tests [JDHW09]. Developers may also use language features such as inheritance or generics to refactor tests.

We believe that retrofitting existing test suites to reduce undesirable clones is valuable. According to Saff, the use of theories reduces the long term maintenance cost for test suites [Saf07]. Moreover, Thummalapenta et al. conclude from an empirical study of existing test suites that parametrization is beneficial—their results indicate that test suites, when retrofitted with parametrization, can detect new defects and provide increased branch coverage [TMX+11]. Test refactoring also generally reduces the brittleness and improves the ease-of-understanding of test code.

However, test refactoring technologies currently require the developer to manually identify opportunities for refactoring, which becomes increasingly difficult as test suite size continues to grow. While writing new tests using test refactoring techniques is often the correct long-term decision, it is not always clear when such techniques apply. Incremental evolution and short-term imperatives may slowly lead to situations where test cloning gets out of control.

The output of our tool—sets of likely test clones—enables developers to regain control, refactor test clones, and improve the quality of their test suites. Furthermore, our work enables the evaluation of novel language features for refactoring tests (and other code), and the results that we have gathered will help with evaluating the usefulness of proposed language features on a realistic set of real-world benchmark suites. There is a rich body of work on clone detection. However, existing clone detection techniques usually treat the code as a sequence of input symbols; they generally do not have a deeper understanding of the code under analysis and can often be foiled by, for instance, changes to the order in which a test computes assertion parameters. Our work leverages insights into the structure of unit tests to identify clones and to give more useful feedback to developers about where clones exist, enabling targeted refactoring efforts. Refactoring yields a test suite with fewer clones. Simpler suites are easier to keep up-to-date as the system under test evolves.

We have formulated a technique for analyzing suites of unit tests and detecting clones. We implemented our technique using the Soot program analysis framework. Our tool analyzes test suites and displays likely-cloned sets of test methods along with the evidence, in terms of specific program fragments, used to draw this inference.

Using our tool, we conducted an empirical study based on a suite of 10 benchmark programs ranging from 8,000 to 246,000 lines of code. Our technique identified from 19% to 70% of test methods as clones in our benchmark suite. We manually inspected 191 randomly sampled clones from these tests and found that our recommendations were often clones, amenable to refactoring.

Our primary contributions are:

- a technique for identifying test-level clones in existing test suites based on assertion fingerprints;
- an implementation of that technique; and
- an empirical study of a significant suite of benchmark programs using our technique.

Our technique computes, for each test method, an ordered set of fingerprinted assertions. It then identifies sets of tests with identical ordered sets of assertions, and filters out likely false negatives. Finally, our display tool shows the computed clone sets to the developer. In our experience, our tool reports many refactorable clones and few false negatives.

References

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, 2009.

[Saf07] David Saff. Theory-infected: Or how I learned to stop worrying and love universal quantifica tion. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming syst ems and applications companion*, OOPSLA '07, pages 846–847, 2007.

[TMX+11] Suresh Thummalapenta, Madhuri R. Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Retrofitting unit tests for parameterized unit testing. In *Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software*, FASE'11/ETAPS'11, pages 294–309, 2011.

[TS05] Nikolai Tillmann and Wolfram Schulte. 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*, ESEC/FSE-13, pages 253–262, 2005.