HoliCoW: Automatically Breaking Team-based Software Projects to Motivate Student Testing

Peng Zhang, Jules White, and Douglas C. Schmidt
Vanderbilt University, Nashville, TN
{peng.zhang,jules.white,d.schmidt}@vanderbilt.edu

ABSTRACT

Intensive testing is often applied by professional software engineers to assure the quality of enterprise information technology (IT) systems. For example, Netflix's Simian Army consists of services that generate various types of failures, detect abnormal conditions, and test the ability of cloud-based enterprise IT software to survive them. Although software engineering students should be taught these types of rigorous testing techniques, it often hard to motivate students to produce high-quality test suites for their assignments since classroom environments lack the harsh outcomes of unexpected system failures.

This paper provides two contributions to work on strengthening coding and testing skills of software engineering students by aligning educational environment more closely with real-world industries. First, we describe the Holistic Code-Wrecker (HoliCoW), which is our testing method and tool that simulates production environments through forced logical error injections into student projects. The modified versions are then run against regression tests written by students and the test results are analyzed to determine the robustness of original software. Second, this paper describes preliminary results from our ongoing experience applying HoliCoW to Software Engineering project courses at Vanderbilt University, where the tool is used to automatically evaluate student software project submissions to determine whether regression tests they define detect errors injected into their code.

Keywords

Software engineering education, software testing, software error injection

1. INTRODUCTION

Motivating the need for testing in enterprise IT systems. Ensuring system availability and reliability is a top priority for software engineers. Enterprise IT systems, such as Amazon's cloud infrastructure and Paypal's payment infrastructure, have substantial financial impacts when they go down. Traditional approaches engineers use to architect robust highly available and reliable industrial systems are based on intensive testing, such as unit, integration, and user acceptance testing. These testing methods are critical to maintaining availability and reliability in enterprise IT systems because they help identify software weaknesses in advance, forcing application developers to continuously improve the code until it reaches an acceptable level of resilient.

Motivating the need for students to focus on testing. When educating software engineering students, it is important to teach them rigorous testing techniques and train them to write high quality tests that cover as much code as possible. The software students typically produce for their assignments, however, does not confront the same harsh environments encountered by software engineers developing production enterprise IT systems, which must handle unanticipated system failures, teams of developers making concurrent modifications, and many users providing input. As a result, students are rarely motivated to produce high-quality test suites, which means that they enter the workforce with inadequate knowledge and skills to write high quality software.

Promising solution approach $\rightarrow$ Forced injection of logical errors into student software. An approach being applied to enhance enterprise IT system resiliency is to intentionally inject errors during the development phase, thereby forcing software engineers to account for potential system failures. The Simian Army [5] created by Netflix is a suite of DevOps tools that (semi-)randomly terminate virtual machines and perform other erroneous activities to simulate hardware component failures and force developers to create effective fault tolerance and recovery mechanisms for cloud-based software. By ensuring software engineers encounter these failures during development they are motivated to account for them deliberately, rather than viewing failures as remote possibilities.

We have adapted the Simian Army approach into our *Principles of Software Engineering* course, which is taught to 40 seniors and undergraduates each year at Vanderbilt University. Our approach is called the *Holistic Code-Wrecker* (HoliCow), which is a method and tool that motivates students to improve their test suites by exposing the software in their software projects to harsh simulated environments. Prior approaches for the Simian Army focus on creating (semi-)random failures in the environment the software depends on, but do not (necessarily) focus on viewing fellow software developers as part of the Simian Army. In team-based class projects, however, teammates often make logical errors and misuse methods written by other team members, which motivates the need for creating test suites that ensure the project software is resilient and thoroughly tested.

HoliCoW simulates the process of software developers making mistakes and *automatically inserts carefully-crafted logical changes into student code via an iterative series of error injects.* Each logical change (such as changing a "<=" conditional statement to "<") is designed to produce no compilation errors, but instead create a logically modified program that is incorrect with respect to the original specification. The test suite produced by the student(s) for the software project is then run to see if the injected logic errors are detected. If the tests fail, the students have sufficiently covered that potential logical error. If the tests pass, the test suite is not sufficiently robust.

The remainder of this paper is organized as follows: Section 2 discusses background and related work; Section 3 gives an overview of HoliCoW and describes how we are applying it to motivate our software engineering students to produce better test suites and higher quality software; and Section 4 presents concluding remarks and outlines future work.

2. BACKGROUND & RELATED WORK

The Simian Army and regression testing are the inspirations behind our research on HoliCoW. This section briefly explores the key points of each and how they play different roles in strengthening systems.

2.1 Simian Army

The Simian Army [5] is a suite of DevOps tools developed and used by the popular subscription service Netflix increase the resilience of their cloud-based systems by motivating their software engineering teams to prepare for inevitable runtime anomalies [3]. These tools randomly induce failures to Netflix's development environments in real time, ranging from shutting down a single hardware component (Chaos Monkey) to bringing down an entire region of multiple data centers (Chaos Kong). The goal of the Simian Army is to enhance user experience by increasing reliability and availability of cloud-based applications and services since *actual* failures in production deployments of the Netflix cloud are handled by applications and system infrastructure in the same manner as cases of the injected failures since developers have already encountered and remedied them throughout the software system lifecycle.

2.2 Regression testing

Regression tests allow developers to discover bugs and errors in early phases of the software lifecycle, when it is less costly to make corrections. Each unit test in a regression test suite targets a particular method, given which a set of desired inputs to produce corresponding output(s). These tests should cover all logical pieces of the code, including but not limited to sequential statements, branching, and looping conditionals. Generally, the more comprehensive the unit tests are in a regression test suite, the more resilient the application is to runtime errors [4]. Test coverage tools are often used in cases of larger code bases to report the percentage of logic covered by the tests. Most coverage tools, such as Istanbul (`github.com/gotwarlost/istanbul`), can also help identify which lines of code are not covered by unit tests.

The primary difference between the usage of Simian Army and regression test suites is that the former utilizes external components (such as the hardware and/or middleware) to target weaknesses in cloud-based systems, where as regression testing focuses more on the internal soundness of a software by predicting possible invalid user inputs and operations. Real-world engineers embrace failures exposed by both techniques and thrive on the lessons learned from discovering the weaknesses in their systems [1]. Education environments, however, often fail to motivate the utility of both these techniques since classroom settings are traditionally not well-equipped to emulate a production software environment due to the following reasons:

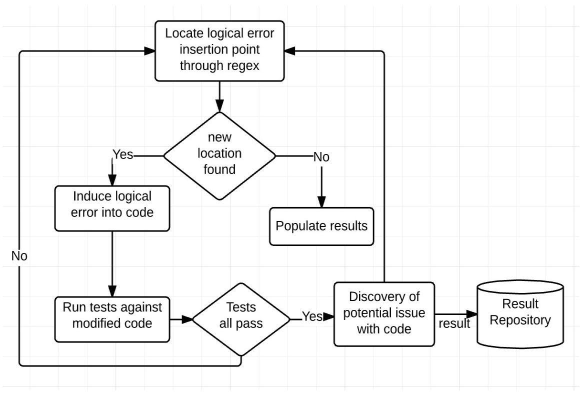
- Students do not encounter users who subject software to a wide range of use-cases (both anticipated and unanticipated)
- Students often work on isolated projects that lack other team members who accidentally misuse or inject errors into the source code.
- Many students have not been exposed to production software experience, which makes it hard to predict possible cases that could crash their solutoins.
- Instead of utilizing code coverage tools as proper leverage, students may misapply them by performing partial testing, yet still receive high-percent code coverage reports [2].

Although schools are increasingly involving software engineering students in more realistic projects, the emphasis is typically on creating apps rather than systematically producing effective test cases.

3. OVERVIEW OF THE HOLISTIC CODE-WRECKER (HOLICOW)

As described in Section 1, the goal of HoliCoW is to motivate students to write more comprehensive test suites by placing their code in a harsh production environment that is representative of a professional development team with actual users. The primary technique employed by the HoliCoW tool is to inject logical errors that produce no compilation errors into the code, but can help pinpoint inadequacies in test suites. The workflow of the tool is shown in Figure 3 and summarized below:

- Locate each possible logical error injection point in the source code written by students, including (but not limited to) variable intialization, branching statements, and looping iterations.
- Randomly modify vulnerable statements found above, *e.g.*, by changing variable values or tweaking branching and/or looping conditions. These modifiations do not affect the compilation of the software, just the correctness of its runtime execution.
- Run the same original unit tests for each modified version to examine the test results. Test suites that are well-written should continually detect fails as a result of the errors injected into the source code.



Locate logical error insertion point through regex
new location found
Yes
No
Induce logical error into code
Populate results
No
Run tests against modified code
Tests all pass
Yes
Discovery of potential issue with code
result
Result Repository

Figure 1: The Workflow of HoliCoW

- After errors are inserted and tests run, output results pinpointing the changes made that do not result in any test failing.

The remainder of this section describes the structure and functionality of HoliCoW and then explains how HoliCoW is being applied in our *Principles of Software Engineering* course at Vanderbilt University.

3.1 The Structure & Functionality of HoliCoW

HoliCoW is written in JavaScript and NodeJS and provides a command-line tool that uses regular expressions to match given source code against a set of code patterns deemed vulnerable to logical errors. After finding such matches, the tool automatically modifies these patterns to create new versions that introduce off-by-1 errors and runs the given test suite, observing whether these errors are captured as expected.

Table 1 shows a subset of the changes that applies to student code that was written in NodeJS as simply a function that displays the amount of change a cashier should return to the buyer, *e.g.* after a transaction is made at a grocery store. The input to the function are price of the purchased item and two separate arrays containing the amount of each type of bills and coins in the cash drawer and the payment from the buyer, respectively. The function is expected to produce an array output consisting of each type and amount of change returned to the buyer or display appropriate messages to the cashier.

Table 1: Subset of Statements HoliCoW Altered

Original Code	Modified Version
if (totalCash[1] < change)	if (totalCash[1] <= change)
if (currentChange > 0)	if (currentChange >= 0)
if (parseInt(change*100) < 0)	if (parseInt(change*100) <= 0)

For this sample program, HoliCoW automatically applies minor changes to the if conditions in the original source code and runs the original unit tests, written in MochaJS, against each modified version.[1] Istanbul, a code coverage tools reports 100% coverage on all unit tests, but all unit tests pass when run against the third modified version.

Upon further examining the original source code, the change in version v3 is more efficient because it breaks out of a for loop containing the if statement at the most opportune time. This example shows a simple proof-of concept to illustrate how HoliCoW combines unit testing and code coverage and provides interesting insights into correctness of code, as well as the coding style of a software developer.

[1]Due to limited space, the unit test results are not shown here.

HoliCoW is a complementary approach to other techniques that can also be used to measure test suite quality, such as code coverage. As part of the HoliCoW testing process, test coverage is also measured to provide students as additional feedback on their code. The possible outcomes are categorized into the three distinct classes described below.

A. 100% code coverage and no modified versions pass all unit tests.

Class A is the most desirable outcome of an application because the unit tests cover all the logical pieces of the source code under test, and modified versions of well-written code are expected to fail the tests, therefore, source and test files falling into this category should be given more confidence in its correctness.

B. Less than 100% code coverage and some modified versions pass all unit tests.

Although a less desirable outcome, Class B is the most common case in student code. HoliCoW thus gives students better intuition into how to improve the unit tests to cover all logic in the source code. In particular, the logical errors that HoliCoW injected and passed the test cases provide direct examples of how a simple programming mistake by themselves or another developer could break their code. These logical flaws identify specific section(s) of code that were neglected in the test suite.

C. 100% code coverage and some modified versions pass all unit tests.

Class C is the most interesting category and also inspires future research experiments. If the test suite covers 100% of the software logic, but HoliCoW is still able to inject a logical flaw that passes the test suite, then some fundamental issues or questions must be raised regarding the code, *e.g.*, are there still potential bugs in the original software uncaptured by the test suite? If not, then it may identify some underlying issue centering on code efficiency and effectiveness, which deserves more attention than writing test cases for an application in educating software engineers. A series of questions can be asked to help students discover the weakness in their coding style, *e.g.*. Could the original code be written more efficiently through delayed variable declaration or precise branching/looping conditions or is there redundant or extraneous variables or logic?

3.2 Observations from Applying HoliCoW in Practice

HoliCoW is employed in our student software projects as part of the assignment specification. Students are told that their assignment will be run through the HoliCoW code modification tool and automatically have 1,000s of logical flaws injected into their code. Initial results (which we are currently rigorously and statistically validating) from applying HoliCoW to software projects in our *Principles of Software Engineering* course are summarized below.

1. Students become much more aware and concerned about testing when HoliCoW is applied.

Our initial analysis indicates that student assignments produced with this process typically have 2-3 times more tests than assignments that do not employ HoliCoW.[2] An interesting aspect of assignments employing HoliCoW is that students quickly become aware of what easy and hard test is, which in itself can be a valuable feedback to them. One of the most common questions in office hours is how user interface or network code can be tested effectively. Prior to applying HoliCoW, students would not typically consider testing this type of code. When faced with random changes being injected by HoliCoW, they become much more concerned about how they are going to provide end-to-end test coverage of their application.

2. Students tend to focus on integration testing when HoliCoW is used.

In assignments *without* HoliCoW (but where part of the assignment grade is based on test suite quality), students focus more heavily on large numbers of unit tests. This outcome may result because students believe these tests appear more substantial and indicative of a large amount of work. Units are often not thoughtfully produced, however, and have limited effectiveness in detecting bugs. In assignments with HoliCoW, conversely, students tend to focus on integration tests and are much more rigorous and detail oriented with regard to ensuring input and output coverage through boundary value analysis.

4. LESSONS LEARNED & FUTURE WORK

This paper described how HoliCoW's "random bug injection" approach modifies a piece of code in an application and runs the same regression tests against the modified code. This approach complements and enhances conventional unit testing methods. In particular, by combining a code coverage tool, HoliCoW helps make software engineering education more comprehensive by motivating students to write additional end-to-end tests, training them think more holistically when testing than simply achieving high code coverage, and increasing their awareness of the importance of code quality.

The following are the key lessons we learned thus far from developing and applying HoliCow to our *Principles of Software Engineering* course at Vanderbilt University.

1. Limitations of existing approaches in an educational environment.

Existing approaches like Netflix's Simian Army and regression testing are helpful in improving the availability and reliability of enterprise IT systems. In software engineering education, however, these approaches alone often do not motivate students to follow best practices for quality assurance. Students are often not motivated to test their code because they do not have real users of their applications. However, using a random bug injection approach shows promise in improving test production and coverage by students.

2. Advantages gained from employing HoliCoW.

[2]Throughout the discussions of initial results below we caution readers that the HoliCoW research is ongoing and that these results are preliminary.

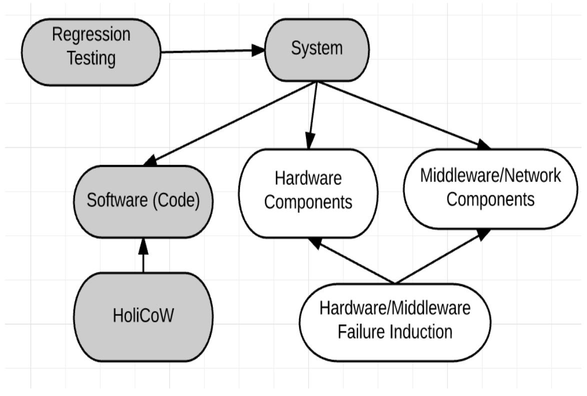

Regression Testing
System
Software (Code)
Hardware Components
Middleware/Network Components
HoliCoW
Hardware/Middleware Failure Induction

Figure 2: The Role of HoliCoW in Software-reliant Systems.

Informing students in advance that a tool will be used to randomly change their code motivates them to write better unit tests and trains them to pay attention to software development fundamentals so they their code passes their tests. An interesting emerging result is that students tend to focus on integration testing when random bug injection is applied rather than more simplistic unit testing.

Our future work focuses on completing detailed experimental validation of the informally observed outcomes discussed in Section 3.2. We also studying interesting relations between different logical modifications that commonly produce results falling into Class C discussed above and conducting experiments to capture metrics relating number of lines of code and percentage of passed test run against modified code. We are extending the HoliCoW tool and integrating it into existing code coverage tools to collect more data for further analyses. Finally, we are conducting experiments to see how directly providing HoliCoW to students enables continuous feedback during their assignments, thereby impacting their test suite and code quality.

5. REFERENCES

[1] T. Limoncelli, J. Robbins, K. Krishnan, and J. Allspaw. Resilience engineering: learning to embrace failure. *Communications of the ACM*, 55(11):40–47, 2012.

[2] B. Marick. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*, pages 16–18, 1999.

[3] H. Maruyama, R. Legaspi, K. Minami, and Y. Yamagata. General resilience: taxonomy and strategies. In *Green Energy for Sustainable Development (ICUE), 2014 International Conference and Utility Exhibition on*, pages 1–8. IEEE, 2014.

[4] R. Osherove. *The art of unit testing*. MITP-Verlags GmbH & Co. KG, 2015.

[5] A. Tseitlin. The antifragile organization. *Communications of the ACM*, 56(8):40–44, 2013.