Problem: Software protects our information, powers our radiation machines, and navigates our airplanes. With adequate testing, engineers can detect bugs in these systems early and prevent disasters. However, testing software systems with large webs of dependencies is difficult. External resources may be fragile or computationally expensive, or they may impact databases with real customer information. It is also unclear whether failed tests are a result of failures in the code itself or in its dependencies. To address these issues, programmers introduce *mock objects* to isolate the code under test. A mock object replaces a real object in a test and imitates the object by responding to input with programmer-defined outputs. A mock database, for example, programmatically handles updates from a test without impacting customer data.

Tests with mock objects are difficult to write and to maintain. In industry, it is common to use mocking frameworks to write mocks. In Java, for example, programmers may use EasyMock, Mockito, or PowerMock. In these frameworks, programmers need to specify every behavior as an input-output pair, and proper execution of the tests depends on the order in which these behaviors are specified. Often, the cases are so numerous that it would be less work to rewrite the underlying method, which is one of the problems that mocking frameworks are trying to solve to begin with.

Existing work in *dynamic test factoring* runs on system tests, captures inputs and outputs, and uses them to introduce mock objects automatically [3, 4, 5]. An analogous dynamic analysis could run the unit tests against the underlying code at first, capture information, and use it to generate mocks. However, **interacting with the underlying code even once to generate mocks can impact customer data in production systems.** Furthermore, it can be expensive, and bugs in the underlying code can impact the generated mocks.

A static approach would solve these problems by never running the underlying code to begin with. **I propose using *static test factoring* to generate mock objects.** The *static analysis* will operate under the assumption that programmers write their own unit tests and specify which objects they want to be mocked. It will examine the code under test and the tests themselves to determine a set of mock inputs and outputs that minimizes the burden of specifying mock inputs and outputs on the programmer.

Intellectual Merit – Research: In their work on automatic test generation, Islam and Csallner dismiss static approaches, noting that they are often incomplete, undecidable, or overapproximate [3]. However, it is possible to circumvent these issues by limiting the scope of the problem. I will focus on minimizing the amount of work required for programmers to write mocks rather than on generating perfect and complete mocks. This makes a static approach a viable option.

One central difficulty in mock generation, whether static or dynamic, is in **generating mocks that do not mask bugs**. I will iterate on the initial tool by combining static analysis techniques with *program synthesis* to solve this problem. Synthesis automatically generates programs from specifications. The act of writing an incomplete test and having a tool fill in the blanks lends itself naturally to synthesis. In effect, by writing unit tests, the programmer is partially specifying the expected behavior.

Existing work on synthesizing mocks does not lessen the burden on the programmer, but instead moves it. Synthiamock, for example, generates mocks from programmer-designed contracts on interfaces[1], but has high annotation burden. By using the tests themselves as constraints along with information about the code from the static analysis, it may be possible to generate input-output pairs that do not mask bugs without requiring any additional work from the programmer.

Intellectual Merit – Personal Background: I have a strong background in programming languages and experience with designing and implementing static analyses. My background as a mathematics researcher is complementary, as static analysis and program synthesis are both heavily based on mathematical concepts. I have a hands-on understanding of the impacts of these problems after spending three years as a software engineer in industry. I can draw upon my past coworkers to understand the impacts of my work, and I can draw upon prominent researchers at UW to assess current work.

Broader Impacts: The proposed research will improve test quality. This will prevent bugs that may otherwise cause a compromise in customer information or customer safety. To ease the process of writing mocks, it is common practice for software engineers to define a set of common mock arguments and responses in a single block which runs before each test. Defining this block is the bulk of the work, and can take as long as writing the code under test itself. The resulting mocks break easily when code is changed or when tests are changed, so that the cost of maintenance is similarly high. Engineers are expected to test their code and at the same time to drive new features on tight timelines. These expectations are unrealistic when the tests are so time-consuming to write. Test quality is often compromised, and bugs go undetected.

The proposed research will solve these problems by making it easier for engineers to write tests for real systems with dependencies. Engineers will be able to run the tool every time they make changes that impact test behavior, so that they will not have to spend time on test design and maintenance. This will encourage engineers to dedicate more effort to the interesting test cases that benefit from human attention and creativity, rather than on manual efforts that can be automated. This will result in an increase in test and code quality, which directly translates to better software. **Better software can protect customer information and save lives.**

Engineers will spend less time writing tests and fixing bugs, and more time driving new features. Engineers will also have more time to dedicate to mentorship. Just one extra hour per week is enough time to take on a new mentorship role for a younger engineer. During my time at Amazon, I worked as a mentor for several engineers and had two mentors myself. I grew as an engineer with the help of my mentors. We were often swamped with development and testing work, however, and could not always dedicate as much time to each other as we wanted to.

Research Plan: I will carry out the proposed research over three years. I will first design, implement and test a static analysis to generate mocks. I will then build on this by designing constraints to synthesize mocks that do not mask bugs and incorporating the two approaches into a single tool. I will test the tool on real programs and compare it to existing mock generation approaches for mock quality, programmer effort, and human readability. I will build on the design until it excels in all three realms.

---

1. Galler, S. J., Weiglhofer, M., and Wotawa, F. Synthesize It: From Design by Contract to Meaningful Test Input Data. In *SEFM '10: 8th IEEE International Conference on Software Engineering and Formal Methods*, pages 286-295.
2. Glassner, D. Test Factoring with amock: Generating Readable Unit Tests from System Tests. Masters Thesis, MIT, 2007.
3. Islam, M., and Csallner, C. Generating Test Cases for Programs that Are Coded against Interfaces and Annotations. *ACM Transactions on Software Engineering and Methodology*: 23(3), Article 21, 38 pages.
4. Saff, D., Artzi, S., Perkins, J. H., and Ernst, M. Automatic Test Factoring for Java. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, pages 114-123.
5. Saff, D. and Ernst, M. Mock Object Creation for Test Factoring. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pages 49-51.