

Automatic generation of transparent interaction-based software unit tests from system tests

MIT Course VI Thesis Proposal: David Glasser

January 16, 2007

Abstract

Automated unit tests are essential for the construction of reliable software, but writing them can be tedious. Several tools exist which attempt to automatically generate small tests from large system tests, but none of them can generate “transparent” tests which look and run like hand-written unit tests. I propose to build a system, **amock**, which generates transparent interaction-based JUnit tests based on dynamic analysis of a system test.

Introduction

Automated testing is essential for the construction of reliable software. Computer systems are complex enough that even small changes can have far-reaching consequences. It is important to ensure both that changes do not affect the overall operation of the system and that internal interfaces continue to function as expected; thus both macroscopic *system tests* and microscopic *unit tests* are a key part of any project. However, writing tests can be one of the most tedious parts of the software development process. While most software engineers recognize the importance of automated testing, many projects fail to achieve a desirable level of testing.

Unit tests can be more useful than system tests for two reasons. First, if a developer changes one small part of a program, running the unit tests which exercise just that part can be more efficient than running an entire system test. Secondly, system test failures often only reveal whether or not the test passed; when a system test starts to fail, it is often difficult to find which subsystem the error originated in. Unit tests are more focused, and so when a unit test fails, it is generally easier to tell which module or even method is responsible for the failure.

Because writing unit tests is important but tedious, the possibility of automating any part of the test creation process is very attractive. Many different approaches to automatic unit test generation have been studied. One promising direction is “test slicing” — transforming system tests into suites of unit tests. It is relatively easy to write a few system tests for a working program — simply run the program with some specified input and check that the output is as expected. Writing unit tests is much more time consuming, since a program typically has a very large number of individual units (classes, methods, etc.) which would profit from unit testing. The key insight of test slicing is that a system test (which is easy to create) exercises the units of a program in a “typical” way. A program that observes the effects of a system test on just one submodule can generate a test that repeats those effects without needing to run the rest of the system test. Thus, we should be able to automatically generate a suite of unit tests from a single system test.

Any automated test generation system will generate tests that are too tightly bound to a current implementation and whose failures thus do not indicate real errors. Therefore, it is essential that generated tests can be read by the developers that will be running them, and they should preferably look similar to tests that a developer would write by hand. Additionally, a test that is too brittle might still be salvageable if the developer can fix it by hand, perhaps by relaxing an expectation constraint. While several test slicing projects exist for Java [18, 7], none of them integrates well into a standard JUnit-based testing system. Instead of just being a familiar sequence of Java methods and JUnit assertions, they use their own custom formats for storing and replaying tests in special instrumented runtime environments. This simplifies the tool’s task by allowing more direct and precise setup than is achievable with ordinary code execution, but the tests they create are not *transparent*: they do not look like standard JUnit tests that developers already understand. The tests they generate can only be used or discarded, not improved. I propose to build a system, *amock*, which generates transparent interaction-based tests for Java systems based on dynamic analysis of system tests.

System, Stated-Based, and Interaction-Based Tests

While all sorts of software tests can be valuable, *amock* will generate interaction-based unit tests by observing system tests. The two major classes of automated tests are *system tests* and *unit tests*; unit tests can generally be subdivided into *state-based* and *interaction-based* tests. Interaction-based unit tests differ from system tests by being more focused, and from state-based unit tests by observing the communication patterns between objects instead of on their final states. This section describes the differences between these types

```

1 #!/bin/sh
2
3 java -jar my-app.jar inputData.txt >outputData.txt
4 diff outputData.txt expectedData.txt

```

Figure 1: A sample system test.

```

1 public class CookieMonsterTest extends TestCase {
2     public static void testCookieEating() {
3         // Set up fixtures.
4         CookieJar j = new CookieJar();
5         Cookie oatmeal = new OatmealCookie();
6         j.add(oatmeal);
7         Cookie chocolate = new ChocolateCookie();
8         j.add(chocolate);
9
10        // Set up primary object.
11        CookieMonster cm = new CookieMonster();
12
13        // Call a method and assert its return value.
14        assertEquals(2, cm.eatAllCookies(j),
15                    "cookie_monster_found_and_ate_two_cookies");
16
17        // Check the state of the fixtures to make sure they're what
18        // is expected.
19        assertTrue(j.isEmpty());
20        assertTrue(oatmeal.wasEaten());
21        assertTrue(chocolate.wasEaten());
22    }
23 }

```

Figure 2: A sample state-based unit test using JUnit.

of tests.

System tests, also known as *end-to-end tests* or *acceptance tests*, consist of running an entire program or large module with specified inputs and expected outputs; a typical system test is shown in Figure 1. System tests generally exercise programs in ways similar to their actual operation, and thus can confirm that typical uses work as expected. As long as a program’s execution can be automated at all, it is generally easy to write a few system tests by hand for any high-level feature. However, when system tests fail, it can be difficult to discover what part of the program is responsible. By the time a fault causes a top-level failure in a system test, it may have propagated through several layers of abstraction and may appear very different from its original cause. Additionally, if a program needs access to resources such as databases or other networked computers, these resources are also needed in order to run the system test. Thus, while developers should be able to create some system tests by hand without much trouble, it would be better to also have tests which can catch a fault closer to its source.

Unit tests complement system tests by focusing on smaller modules, such as single classes or methods. A typical unit test in an object-oriented system consists of some setup followed by a small number of method calls on a small number of objects, with assertions inserted to check the results of the method calls. Often,

```

1 public class CookieMonsterTest extends MockObjectTestCase {
2     public static void testCookieEating {
3         // Set up mocks.
4         Mock j = mock(CookieJar.class);
5         Mock oatmeal = mock(Cookie.class);
6         Mock chocolate = mock(Cookie.class);
7
8         // Set up primary object.
9         CookieMonster cm = new CookieMonster();
10
11        // Set up expectations and stubs for the mocks.
12        j.expects(exactly(3))
13            .method("getACookie")
14            .withNoArguments()
15            .will(onConsecutiveCalls(returnValue(oatmeal.proxy()),
16                                    returnValue(chocolate.proxy()),
17                                    returnValue(null)));
18
19        oatmeal.expects(once()).method("eat")
20        chocolate.expects(once()).method("eat");
21
22        // Call a method and assert its return value.
23        assertEquals(2, cm.eatAllCookies(j),
24                    "cookie_monster_found_and_ate_two_cookies");
25
26        // Mock object expectations are verified automatically.
27    }
28 }

```

Figure 3: A sample interaction-based unit test using JUnit with jMock.

resources such as databases are replaced with hand-written “stub” implementations which allow the tests to be run without needing to set up actual instances of the resources. Most modern programming environments have a standard framework for creating unit tests, such as Java’s JUnit [3], which provides developers in different organizations with a common framework for running unit tests.

Unit tests can generally be divided into *state-based* and *interaction-based* tests [9]. State-based tests create the primary object that’s under test, as well as any other objects needed to run the test (some of which many be stubs); these objects are often called *fixtures*. The test calls whatever methods are under test, and then examines the primary and fixture objects to make sure that they are in the expected state. State-based unit tests ignore what the tested methods do during their execution, as long as the program does not crash; they just check to make sure that the eventual results of the methods are appropriate. Figure 2 shows a typical state-based unit test.

Interaction-based unit tests can be thought of as a direct interpretation of the object-oriented paradigm: the primary elements of a program are the objects and the messages they pass to each other, and so tests should verify that the correct messages are passed. Instead of focusing on the state of objects after a tested method is called, interaction-based tests pay attention to the method calls made by the tested method. For example, in order to test the transfer operation in a banking system, a state-based test would check that the balances of the affected account had changed after calling the transfer method; an interaction-based

test which check that the transfer method passes messages to the affected accounts telling them to add and subtract money from their balances. A typical interaction-based unit test is shown in Figure 3. Interaction-based tests thus consider the way that an object interacts with its environment to be the most important part of its contract.

Interaction-based tests are often implemented using *mock objects*[16]. Mock objects are a special kind of fixture or stub which have a way of verifying that an expected pattern of messages are passed to them during a test. Developers can use *mock-object generation libraries* such as jMock [13, 10] to create mock objects and specify how they should react to messages and what messages they should expect.

It can be difficult to write clean interaction-based tests for all Java programs; modules with complicated and ad hoc ways of interacting with their environment require equally complicated mock objects. In fact, some proponents of interaction-based testing claim that its strongest advantage is encouraging clean OO design, as it works best with programs that follow OO design guidelines such as the “tell, don’t ask” principle and the “Law of Demeter” [12, 15, 11].

Previous Work in Test Slicing

An ideal automated test suite fully “covers” the entire system with automated tests that verify every important and documented external and internal interface. However, writing such tests can be a time-consuming and tedious process. There is a wealth of literature on automatically generating software tests. The techniques range from random testing [19, 17, 5] to systematic static analyses [6, 20] to model-based dynamic analyses [2]. Most of the work in test generation focuses on creating state-based tests. Two projects exist which attempt a “test slicing” strategy of turning dynamic traces of large system tests into smaller unit tests: an interaction-based system called “test factoring” [18] and a state-based system called “test carving” [7].

The main goal of test factoring [18] is allowing developers to run a slow system test much more efficiently when only a small part of the system has changed. Test factoring creates a transcript of a long system test or other program execution, and then plays it back in a special instrumented Java run-time environment where all objects other than those of the class under test are mocks following the transcript. Method calls that only involve classes not under test are essentially skipped, leading to a much faster execution of the original test suite which only exercises code from one class. If the class under test attempts to make different method calls to the rest of the system than it did during the original execution, the replay stops and tells the

user that the full system test should be run instead; otherwise, the factored test succeeds or fails according to whether the program that it is replaying succeeds or fails.

With test factoring, we can make the strong claim that if a factored test for the only class whose code has changed passes, then the system test would have passed as well. However, many benign changes to the class can cause test factoring to fail to replay the test. For example, the method under test could call external methods in a slightly different order or with slightly different parameters; test factoring will consider this to be too different to continue the replay, but a human could determine that both versions are acceptable. A developer cannot easily take a transcript made by test factoring and make it less brittle by relaxing these constraints; tests are recorded in a transcript which is not meant for human consumption. Additionally, test factoring only slices up the program state based on class or package names, not based on time or individual object lifetimes: the generated tests consist of replaying an entire system test on the target class. Thus, even if the tests were human-readable, they would be very long; even if a typical use of an instance of the class under test only involves a few method calls, each test includes all of the method calls ever made on any object of that class. Finally, test factoring relies on the ability to instrument all classes (including the JDK system libraries) even just to replay the tests, which makes it non-trivial to integrate into a pre-existing unit testing process.

In test carving [7], during the execution of a long system test, all reachable objects are serializing to disk frequently. Pieces of the long test can then be independently “played back” by loading the state before an action, executing that action, and comparing the actual post-state to the serialized post-state. This method is fundamentally state-based, and relies strongly on the internal data structures of the objects not changing significantly. Test carving produces tests which only work in the context of their custom serialization framework — carved tests look nothing like tests that a programmer would have designed by hand, and it is unclear how much information a developer can get about why a carved test failed.

amock: Generating Transparent Interaction-Based Tests

I propose to create a system, **amock**, which automatically generates transparent interaction-based unit tests for Java software based on dynamic analysis of a system test. By *transparent*, I mean that the generated tests will not rely on a custom runtime apparatus developed as a part of the project; while the generation process may require special infrastructure, the generated tests must rely only on industry-standard testing tools such as JUnit [3, 14] and jMock [10, 13].

```

1 public class CookieMonster {
2     public void eatAllCookies(CookieJar jar) {
3         Cookie k;
4         for (k = jar.getACookie();
5             k != null;
6             k = jar.getACookie()) {
7             k.eat();
8         }
9     }
10 }
11
12 public class CookieJar {
13     private List<Cookie> myCookies;
14     // ...
15     public Cookie getACookie() {
16         if (myCookies.isEmpty()) {
17             return null;
18         } else {
19             return myCookies.remove(0);
20         }
21     }
22 }
23
24 public class Bakery {
25     public static void main(String[] args) {
26         // ...
27         CookieJar j = new CookieJar();
28         Cookie oatmeal = new OatmealCookie();
29         j.add(oatmeal);
30         loadMoreCookies(j);
31         new CookieMonster().eatAllCookies(j);
32     }
33     private static void loadMoreCookies(CookieJar j) {
34         j.add(new ChocolateCookie());
35     }
36 }

```

Figure 4: A sample subject program; running `amock` on this should produce tests like the one in Figure 3.

`amock` will generate a trace of the execution of a system test using standard Java instrumentation techniques. A separate program will process the trace and create small tests exercising single objects. Each test will create a single object, build mocks for other objects that it interacts with using `jMock`, call methods on it corresponding to the actual messages it received during the trace, and assert that the methods had the expected return values and that the mocks received the expected messages. For example, if `amock` processed the program shown in Figure 4, one of the tests it generates might be the one in Figure 3.

While this concept is similar to test factoring, the results are very different. Most importantly, the generated tests are ordinary Java code and require no special infrastructure other than the standard JUnit and `jMock` libraries; the tests could potentially be similar to actual interaction-based tests that a developer would write by hand. Thus the generated tests can be human-comprehensible, and developers should be able to relax overly brittle tests and otherwise improve them by hand. This is preferable to a system that requires the end user to convert their entire development system to use the new tool and are thus unattractive to developers who may not find the results compelling enough to warrant a drastic change in process. `amock` should enable a single developer to generate tests and add them to the project's standard test suite so that other developers can run them without even knowing that `amock` exists.

Unlike test factoring, which creates one long test for each class or package being focused on, `amock` would create separate tests for different instances of the class; a potential extension would be to analyze multiple generated tests and discard those which are structurally identical, which would cut down on the size of the generated test suite without reducing test coverage. Unlike test carving, `amock`'s interaction-based tests would not be sensitive to changes in the internal data structures of the tested objects and fixtures.

`amock` will potentially be able to interface with other analysis projects in order to generate less brittle tests. For example, tests generated by test factoring expect that the object under test makes exactly the same calls to the rest of the program as it did during the system test. This is important when the calls have side effects, but test factoring will raise a replay exception even when methods without side effects are called in a different order or a different number of times. For example, if during the system test an object under test originally called `book.getName()`, `book.getAuthor()`, and then `book.getName()` again without making any calls with side effects to `book` in between, it should be acceptable for the object to instead call `book.getAuthor()` twice followed by `book.getName()` (assuming that these method have no side effects). The current implementation of test factoring would throw a replay exception if the latter series occurs. `amock` should be able to use a mutability analysis such as Palulu [1] to determine if mocked method calls need to actually be *expected* or merely *recognized*. Similarly, when combining multiple similar tests

into a single version, `amock` could use Daikon[8] to write looser and less brittle constraints on the expected messages passed to the mock objects.

Test factoring chose to instrument all classes for a reason — there are many edge cases in Java, and truly mocking out the rest of the world cannot be done perfectly in a completely transparent fashion. I choose to accept this drawback and realize that `amock` will be unable to adequately mock some behaviors of the system test. For example, types in Java can be declared as either interfaces or classes; interfaces have no instance variables, method implementations, or constructors. `jMock` is most adept at mocking interfaces, because a mock implementation of an interface merely has to provide alternate implementations for the methods listed in the interface definition. `jMock` does support mocking classes using the `cglib` code generation library [4]. However, mocking direct accesses to instance fields and static method calls may require some extensions to `jMock`. Additionally, mock classes must call the actual superclass constructor; thus, whatever happens in a superclass constructor must be executed even by the mock object. Finally, programming styles that include many nested method calls like `company.getDivision().getDepartment().getEmployee().getProject().setName()` require creating mock objects returning nested mock objects, generally making for complicated tests.

Some proponents of interaction-based testing, including several of the authors of `jMock`, argue that this apparent drawback of mock-based testing is actually an advantage[11]. To them, mock-based testing is as much of a design paradigm as a retrospective testing tool. After all, most software engineers consider direct access to the instance variables of other objects to be a poor practice. Long chains of method calls, which are sometimes referred to disparagingly as “train wrecks”, reduce loose coupling by breaking the Law of Demeter [15]. According to these proponents of interaction-based testing, aiming to write clean mock-based tests encourages developers to write clean code which properly separates the relationships between objects across clear and documented interfaces. It is thus quite possible that `amock` will create simple and clean tests for code which follows these guidelines, and will create ugly and complicated tests for code which do not. This is not necessarily a failure for `amock`: after all, it is only an encouragement for developers to structure their code in ways that some consider to be best practices.

Project Plan

I have begun an initial implementation of `amock`. Currently, `amock` can produce a dynamic trace of a system test and generate very limited unit tests exercising specified classes. It can currently generate tests that involve primitive arguments and return values, and has some preliminary support for mock object generation.

The current implementation is relatively modular; some parts are written in Java, and others in Ruby. (It is likely that I will rewrite the Ruby sections back into Java.) I plan to work on the implementation extensively over the December break and IAP; by the beginning of spring term I hope to have an implementation which can generate passing mock-based unit tests which mimic the behavior of the system tests.

I plan the following milestones for the project:

- December 22nd – January 1st: Implement an abstraction for representing the tests themselves. In addition to dealing with low-level details like writing Java statements, this will include abstractions specific to mock-based tests, such as tracking the different mocks used by a test and different expected executions of the same method. This will allow for programmatic (but not yet automatic) creation of interaction-based tests.
- January 7th – 12th: Create a set of small subject programs and tests that should be generated from them. Finish the infrastructure for generating and reading trace files.
- January 15th – 26th: Implement `amock` for the special case that all parameters to all methods invoked on the object under test are declared as interface types or primitives. (Mocking interfaces is simpler than mocking classes, because there is no direct access to field data or implicit superclass constructors.)
- January 27th – February 2nd: Begin to deal with class-valued arguments.
- February 3rd – March 1st: Complete end-to-end system capable of producing some tests (if not necessarily high quality tests) on nontrivial projects.
- March 1st – April 1st: Focus on thesis writing, making tests simpler and more comprehensible, and integrating with other analysis projects.

I hope that the implementation of `amock` (at least with interface types) will be fully functional if not incredibly sophisticated before the beginning of spring term. Over the course of the spring term, I plan to focus on making the generated tests less brittle and redundant, incorporating more sophisticated techniques such as the aforementioned Palulu and Daikon integration; I will lay out more specific milestones for the spring at the beginning of the term. I will also write my Master's Thesis itself during the spring term.

I will evaluate `amock` by running it on several real Java projects of various sizes. Metrics to measure include coverage by generated tests and brittleness of tests (which can be observed by running tests against different versions of the same project). I can also perform case studies to see how comprehensible and modifiable the generated tests actually are.

My biggest concern is that it will prove difficult to adequately generate mock-based tests for real programs even if it works on my own test cases. Specifically, I worry that I will confirm that, as suggested in [11], mock-based testing is more useful as a design strategy than as an ex post facto testing strategy, and that even hand-written interaction-based tests are infeasible for systems that were not designed around them. However, this would actually be a positive conclusion: experimental evidence that interaction-based tests are not useful for legacy projects. While this would be disappointing, I do not think it would make the project worthless.

References

- [1] S. Artzi, M. D. Ernst, D. Glasser, and A. Kiežun. Combined static and dynamic mutability analysis. Technical Report MIT-CSAIL-TR-2006-065, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Sept. 18, 2006.
- [2] S. Artzi, M. D. Ernst, A. Kiežun, C. Pacheco, and J. H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)*, Portland, OR, USA, Oct. 23, 2006.
- [3] K. Beck and E. Gamma. JUnit test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.
- [4] cglib. <http://cglib.sourceforge.net/>.
- [5] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, Sept. 2004.
- [6] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE'05, Proceedings of the 27th International Conference on Software Engineering*, pages 422–431, St. Louis, MO, USA, May 18–20, 2005.
- [7] S. Elbaum, H. N. Chin, M. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the International Symposium Foundations of Software Engineering*. ACM, november 2006.
- [8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Programming*, 2006.

- [9] M. Fowler. Mocks aren't stubs. <http://www.martinfowler.com/articles/mocksArentStubs.html>, 2004.
- [10] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. jMock: supporting responsibility-based design with mock objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 4–5, New York, NY, USA, 2004. ACM Press.
- [11] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. Mock roles, not objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246, New York, NY, USA, 2004. ACM Press.
- [12] A. Hunt and D. Thomas. Tell, don't ask. <http://www.pragmaticprogrammer.com/pp11c/papers/1998.05.html>, 1998.
- [13] jMock. <http://www.jmock.org/>.
- [14] JUnit. <http://www.junit.org>.
- [15] K. J. Lieberherr and I. M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.
- [16] T. Mackinnon, S. Freeman, and P. Craig. Endo-Testing: Unit testing with mock objects, 2000.
- [17] Parasoft Corporation. *Jtest version 4.5*. <http://www.parasoft.com/>.
- [18] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE 2005: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, USA, Nov. 9–11, 2005.
- [19] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP 2006 — Object-Oriented Programming, 20th European Conference*, pages 380–403, Nantes, France, July 5–7, 2006.
- [20] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, Edinburgh, UK, Apr. 4–8, 2005.