Detecting, Isolating and Enforcing Dependencies Among and Within Test Cases

Jonathan Bell
Columbia University
500 West 120th St, MC 0401
New York, NY USA
jbell@cs.columbia.edu

ABSTRACT

Testing stateful applications is challenging, as it can be difficult to identify hidden dependencies on program state. These dependencies may manifest between several test cases, or simply within a single test case. When it's left to developers to document, understand, and respond to these dependencies, a mistake can result in unexpected and invalid test results. Although current testing infrastructure does not currently leverage state dependency information, we argue that it could, and that by doing so testing can be improved. Our results thus far show that by recovering dependencies between test cases and modifying the popular testing framework, JUnit, to utilize this information, we can optimize the testing process, reducing time needed to run tests by 62% on average. Our ongoing work is to apply similar analyses to improve existing state of the art test suite prioritization techniques and state of the art test case generation techniques. This work is advised by Professor Gail Kaiser.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*

General Terms

Reliability, Performance

Keywords

Testing, test dependencies, program analysis

1. INTRODUCTION

When creating unit and integration tests, engineers create scripts that setup the application under test and then feed inputs into units, observing the results. Programs may have explicit inputs (e.g. for a chat server, the message passed to the server), or implicit inputs (e.g. for a chat server, the accumulated state of what users are connected and in what rooms). Software that is stateful, or in other words, software that behaves differently according to what actions have been performed on it in the past, poses many difficulties for software testing due to its abundance of implicit inputs. Even if software is intended to be stateless, when testing it we must still assume that it might be accumulating some state, in order to test that it is indeed stateless.

SIGSOFT/FSE '14, November 16-22, 2014, Hong Kong, China

http://dx.doi.org/10.1145/2635868.2666597

```
//Our code under test.
//File: Example.java
public class Example{
  public void doStuff(int input){
    if(Singleton.getFlag()) crash();
  }
}
//Source code for external library we use,
    which is treated as a black-box
//File: libs/evil.jar
public class Singleton{
  private static int n = 0;
  private static boolean flag = false;
  public static void startBlocking(){
    if(n > 2) flag = true;
    n++;
  }
  public static void stopBlocking(){
    flag = false;
  }
  public static boolean getFlag(){
    return flag;
  }
}
```

Figure 1: An example of a state dependency

We therefore typically rely on *pre-test* functions in our testing procedures to bring the system to an acceptable state (be it a real state, or a simulated "mocked" state) before testing the unit under scrutiny, and *post-test* functions to clean up any accumulated state. Consider the greatly simplified code example shown in Figure 1 consisting of two classes: `Singleton` and `Example`. The method `Example.doStuff` contains a hidden crash, which occurs only when the application is in a specific state: when `Singleton.flag` is set to true. Simply calling the method `doStuff` is insufficient to fully explore the unit: to expose the hidden crash, we would first have to call `Singleton.startBlocking` three times.

If the class `Singleton` is a black box, third party API, then it may be hard for developers to determine the set of actions to exercise all of its states — what we will call a dependency on state within a single test case. However, there could also be a dependency *between* test cases: with this state stored in a `static` field, unless it is explicitly cleaned up, it will remain there even after the test finishes, until the application is shut down. If multiple test cases interact with `Single-`

`ton.startBlocking` or `Singleton.stopBlocking` and are run in the same JVM, then the tests may interfere with each other — what we will call a dependency on state between test cases. Detecting these dependencies by hand is risky, and prior work has shown that incorrectly identifying them can lead to undetected faults, despite the existence of test cases that seem to be correct [12].

Hypothesis: The testing process will be made faster and more effective by modifying testing tools to become aware of otherwise hidden state dependencies.

Once detected, we believe that state dependencies within individual test cases can be used to guide test case generation so as to expose software behavior in otherwise hidden states (hence increasing the efficacy of testing). Most existing test generation tools for object oriented software rely on purely random [13] or fuzzing based techniques [21] to generate method sequences, in an attempt to expose such dependencies.

Sound knowledge of state dependencies *between* tests can be used to reduce the time needed to isolate test cases from each other. The previous mechanism for isolating dependencies between tests in Java required completely restarting the JVM — our approach instead only reinitializes relevant portions of memory that may be involved in such a dependency, speeding up test suite execution by 62% on average [2].

While our prior work focused on *isolating* the dependencies between test cases, consider the case where several test cases are dependent on each other, intentionally left unisolated: in these cases, we may want to *enforce* the dependencies. Test suite prioritization is a useful technique that re-orders the execution of test cases, for example, first executing those test cases most effected by a recent code commit. However, if a dependency is expected to occur between tests, by reordering test cases we may break these dependencies and hence create invalid results. We believe that with sound and precise knowledge of dependencies between tests we can improve state of the art test suite prioritization tools to make them resilient to dependent tests.

2. PRIOR AND RELATED WORK

While we are not aware of any work towards specifically detecting state dependencies between units in the same test, systems that perform automated test generation may still be useful for exposing faults in such software [8, 11, 13, 20]. To test object oriented software specifically, many tools use novel approaches to generate *sequences* of methods to invoke before the actual method under test, in order to bring the system into an interesting state for testing.

For example, Randoop [13] uses a guided-random approach to generate sequences of method calls. Harrold and Rothermel applied data flow testing to object oriented programs in order to guide selection of sequences of method calls [9]. Buy, et al. combined data flow analysis with symbolic execution and automated deduction to produce method invocation sequences that generate new application states [7]. Palus [21] and OCAT [11] use real-world executions of applications to identify real-world method sequence or object states to use to guide input generation. MSeqGen statically mines codebases to detect sequences of method calls used to configure objects' state, and then uses these sequences to automatically generate a pre-test method before using traditional dynamic symbolic execution or random testing to test a given method [16]. Evacon uses an evolutionary testing [17] approach to bring the system to interesting states before using dynamic symbolic execution to examine a given method [10].

Our proposal differs from all such work in that we are searching for more subtle dependencies on internal state than may be exposed by simply executing several methods in a sequence, instead looking at broader and more subtle dependencies that may require very complex method sequences to reproduce.

Test case selection techniques (e.g. those which use algorithms to determine a smaller set of tests to run, or a different order to run them in [19]) have long assumed that test cases are independent of each other. To be sound, such work often relied on the "Controlled Regression Testing Assumption" — assuming essentially that there are no factors that may cause tests to behave differently (other than bugs in the system under test) [14]. Zhang et al. performed a study showing that violations of this assumption could be very difficult for developers to detect and could cause non-trivial consequences, building a tool for detecting dependent tests: DTDetector [20]. DTDetector unfortunately is not very applicable to enhance test suite prioritization, as its running time is significantly longer than the time it would take to simply run the test suite once. We propose eliminating this required assumption, instead preserving soundness by automatically enforcing it. In our recent work on Unit Test Virtualization, we proposed a new mechanism for detecting and isolating dependent tests with significantly lower overhead than the existing mechanism, showing improvements of on average 62% in testing time [2]. Our goal of efficiently isolating tests is similar to that in the automated test generation system, JCrasher [8], although JCrasher's isolation only approximates the JVM's normal behavior.

In addition to our relevant prior work that will be discussed elsewhere in this proposal (Phosphor [1], VMVM [2,3] and Chronicler [4]), we also have several other publications on the unrelated topic of gamification, included here for completeness: [5, 6, 15].

3. APPROACH AND CONTRIBUTIONS

At a high level, we will use a variety of program analysis techniques to detect state dependencies within applications, and then adapt existing testing infrastructure to take advantage of this information. These analysis techniques will include existing analyses such as static control and data flow analysis, dynamic data flow analysis, and symbolic execution. We see several key software testing challenges that this dependency information will help us to overcome, including:

1. *Isolating the system state side-effects of test cases is traditionally very expensive.* However, by knowing the specific dependencies that tests may have on each other, we can significantly reduce the amount of time needed to execute test suites [2].
2. *Executing intentionally dependent tests in isolation is complicated.* When a test is dependent on some other test(s), and is normally expected to be executed in conjunction with these other tests, executing it out-of-order can lead to unpredictable results. By efficiently discovering such dependencies, we can allow test selection techniques to behave soundly, even in the presence of dependencies between test cases.
3. *Creating high quality test cases for stateful software is difficult.* Without knowledge of how components in-

teract, it can be difficult to imagine the combinations of components that must be tested to expose all possible software states. Precise dependency and state information will again help to automatically guide test generation to increase coverage.

3.1 Identifying Dependencies

In our prior work, we identified dependencies between entire test cases in Java through a combined static and dynamic analysis [2]. In the static phase, we soundly identified all possible memory areas that could cause dependency conflicts during execution and inserted guards to support our dynamic analysis. These dependency conflicts would manifest as one test case writing to an area of memory and the second test case reading it (without it being re-initialized). In the dynamic phase, we more precisely identified which of these regions posed an actual conflict, and then eliminated the conflict by re-initializing them to their original state.

While these analyses were sound, they were not completely precise: for our previous purposes, a false positive (i.e., reporting a dependency when one did not exist) did not pose a significant risk: the overhead of reinitializing an extra class was not very high. However, when "enforcing" these dependencies during test case selection processes, a high false positive rate may mean that we will group together many (or all!) of our test cases, essentially defeating the benefits of the test selection process.

We plan to augment these analyses with more precise control and data flow information in order to reduce the false positive rate and to support additional applications (such as the challenges listed above). To begin this process, we have constructed a sound and precise dynamic data flow analysis for Java: Phosphor [1]. Phosphor is the first such analysis that works on commodity JVMs (such as Oracle's HotSpot or OpenJDK's IcedTea), and has low enough overhead to use during the testing cycle (on average 53%). We will use Phosphor to create a log of variables where we observed reads and writes that may have been involved in dependencies. Then, we will combine this log with additional statically mined control flow information in order to determine which tests are guaranteed to be independent.

These techniques are oriented to detecting dependencies between test cases. To begin to classify state dependencies within a single test case as it executes, we have extended Phosphor to track path conditions on variables within the JVM. As variables are created and transformed, Phosphor tracks these relationships, and when branch statements occur, Phosphor records a constraint on the condition. These conditions can therefore be used to succinctly represent the relevant state of the application. We can then feed these constraints into an SMT solver and generate concrete inputs that satisfy these (or perturbed) conditions, allowing us to explore uncovered code. Our preliminary results show somewhat higher overhead from this form of logging, but we are confident that with some optimization it will be sufficiently performant to use when testing.

Using these tools, we will begin to explore how to efficiently detect relevant software state in order to determine what dependencies may exist among components within a given test case. We will begin by considering all data which is stored on the heap and used in decision making as relevant state. We can imagine several refinements to this approach to simplify it: for instance, if we are concerned primarily with configuration state, then perhaps these heap variables must be written only once (and not updated). We are also interested in overlaying a control flow graph above the data flow graph to detect state dependencies between components, by clustering the control flow nodes to represent components.

3.2 Using Dependency Information

As described above, we have identified three key areas to which we will apply this dependency information to improve software testing.

3.2.1 Reducing testing overhead

When testing, we typically make an implicit assumption that the execution of one test case should not effect the execution of another. In some cases, this assumption is enforced through the use of pre-test setup and post-test teardown functions, which attempt to reset the system to a clean state. However, perhaps because it can be difficult to identify exactly what part of the application's state needs to be reset for an individual test case (particularly in the case where third party libraries are used), we found that in the majority of large Java projects, developers enforce this isolation by executing each test case in the context a fresh instance of the application, in its own JVM [2]. The overhead during testing of restarting the JVM for every test case can be tremendous, on average 618% in our study.

Rather than restart the entire JVM for each test case, we leverage dependency information to identify which test cases may be dependent on each other, and in particular, what fields in memory are being shared between test cases. Knowing which fields in memory are causing these conflicts, we can greatly improve testing performance by simply resetting those fields, rather than the entire JVM. We implemented this approach for Java, finding that it reduced the time necessary to run the test suites of 20 popular free open source applications by an average of 62% [2]. Our current approach provides isolation equivalent to restarting the JVM, but does not provide isolation across the filesystem or through database calls. We are currently interested in combining VMVM with a copy-on-write filesystem to begin to provide such isolation.

3.2.2 Executing test cases out-of-order

While our previous work efficiently *isolated* the dependencies between test cases, we now seek to extend this to cases where it is necessary to *enforce* dependencies between test cases. Consider the case where developers do not want to isolate their test cases — there are some dependencies between test cases, and while the developers do not know exactly what they are, they are allowed to exist. While some may argue that dependent tests should be found and removed [20], we have evidence that there are many test suites which do not isolate their test cases, and hence, may be allowing their tests to be dependent.

In our previous study, we found that although most large projects isolate their test cases (71% of the 1,000 free open source projects surveyed that used JUnit isolated their test cases), overall, only 41% of the projects isolate their test cases [2]. However, to be sound, test suite selection techniques rely on the assumption that there is *no* dependency between test cases. To remove this assumption and hence increase the soundness of such techniques when applied to test suites which may have dependencies, we will efficiently identify and enforce the dependencies.

Most approaches that we are considering will require executing the test suite in its entirety once, generating profiling data based on data flow, which we will then analyze to detect dependencies. In contrast, the number of times that a test suite must be executed by existing approaches to detect dependencies is exponentially bounded ($O(n^n)$ for n test cases) [20]. Then, when a dependency is detected between tests B and A, we will enforce that the tests must be run in their originally specified serial order. By combining in-memory dependency tracking with a copy-on-write file system, we can eliminate most of the possibilities for test cases to be dependent on each other (we are still studying mechanisms to detect and prevent cross-network dependencies). In addition to allowing for sound test selection, such a technique would also be useful to allow for massive distribution of test cases across many machines in parallel.

3.2.3 Improving testing coverage

Generating meaningful tests for stateful software is challenging because individual methods can not be simply called in isolation — it's necessary instead to call several methods in sequence in order to ensure that the application is in the correct state before the target method is called. One promising approach to generating useful method call sequences is to record a trace of method calls, infer control dependencies, then fuzz these sequences to generate new executions [21]. However, a limitation of this approach is that generated sequences will be based on existing sequences, rather than directly searching for new sequences. Some recent work has shown the practicality of using data flow def-use pairs to generate tests for such software [18]. We believe that a novel combination of data flow coverage driven test case generation (i.e., based on state dependencies) with symbolic execution may help to improve the state of the art of automatic test generation for stateful, object-oriented software.

4. RESEARCH PROGRESS AND PLAN

We have already developed a coarse-grained analysis for detecting dependencies between components, and used this to significantly speedup the testing process [2]. Towards refining the granularity of this analysis, we have developed a highly accurate, efficient and portable data flow analysis for Java [1], which we have since extended to track path conditions and to solve them for concrete inputs using state-of-the-art constraint solvers. Once we have completed our tool for enforcing dependencies between tests, we will evaluate its soundness and precision in comparison to a state-of-the-art tool [20], as well as its runtime overhead on many free and popular open source applications.

We are currently constructing a system for generating system states using a combination of data flow analysis, symbolic execution, and static analysis. We will evaluate our system by comparing the coverage of tests that we generate to those generated state-of-the-art automatic test suite generators and those written by human developers. We will also compare the fault finding ability of these test suites in a mutation analysis, and in detecting real-world bugs. We will continue to make our best effort to publicly release our artifacts to encourage collaboration and extension (Chronicler, VMVM and Phosphor are all on GitHub).

5. ACKNOWLEDGMENTS

The author is advised by Prof Gail Kaiser, both of whom are members of the Programming Systems Laboratory at Columbia University, funded in part by NSF CCF-1302269, CCF-1161079, NSF CNS-0905246, and NIH U54 CA121852.

6. REFERENCES

[1] J. Bell and G. Kaiser. Phosphor: Illuminating Dynamic Data Flow in the JVM. OOPSLA '14 (To Appear).

[2] J. Bell and G. Kaiser. Unit Test Virtualization with VMVM. ICSE '14.

[3] J. Bell and G. Kaiser. VMVM: Unit Test Virtualization for Java (Formal Tool Demonstration).

[4] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. ICSE '13.

[5] J. Bell, S. Sheth, and G. Kaiser. Secret ninja testing with halo software engineering. In *Proc. of the 4th Int'l Workshop on Social Software Engineering*, 2011.

[6] J. Bell, S. Sheth, and G. Kaiser. A large-scale, longitudinal study of user profiles in world of warcraft. In *Proc. of the 5th Int'l Workshop on Web Intelligence and Communities*, 2013.

[7] U. Buy, A. Orso, and M. Pezze. Automated testing of classes. ISSTA '00.

[8] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[9] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. SIGSOFT '94.

[10] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. ASE '08.

[11] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. Ocat: object capture-based automated testing. ISSTA '10.

[12] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. ESEC/FSE '11.

[13] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. ICSE '07.

[14] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE TSE*, 1996.

[15] S. Sheth, J. Bell, and G. Kaiser. HALO (Highly Addictive, sociaLly Optimized) Software Engineering. In *Proc. of the 1st Int'l Workshop on Games and Software Engineering*, 2011.

[16] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. ESEC/FSE '09.

[17] P. Tonella. Evolutionary testing of classes. ISSTA '04.

[18] M. Vivanti, A. Mis, A. Gorla, and G. Fraser. Search-based data-flow test generation. ISSRE '13.

[19] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, Mar. 2012.

[20] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, M. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. ISSTA '14.

[21] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. ISSTA '11.