Repairing Test Dependence

Wing Lam
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
winglam2@illinois.edu

ABSTRACT

In a test suite, all the tests should be independent: no test should affect another test's result, and running the tests in any order should yield the same test results. The assumption of such test independence is important so that tests behave consistently as designed. However, this critical assumption often does not hold in practice due to test dependence.

Test dependence causes two serious problems: a dependent test may spuriously fail even when the software is correct (a false positive alarm), or it may spuriously pass even when a bug exists in the software (a false negative). Existing approaches to cope with test dependence require tests to be executed in a given order or for each test to be executed in a separate virtual machine. This paper presents an approach that can automatically repair test dependence so that each test in a suite yields the same result regardless of their execution order. At compile time, the approach refactors code under test and test code to eliminate test dependence and prevent spurious test successes or failures.

We develop a prototype of our approach to handle one of the most common causes of test dependence and evaluate the prototype on five subject programs. In our experimental evaluation, our prototype is capable of eliminating up to 12.5% of the test dependence in the subject programs.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Test dependence, Refactoring, Testing, Repairing, False alarm

1. RESEARCH PROBLEM

Test dependence causes inconsistent test result when tests are run in different execution orders. Test dependence has two serious consequences:

```
static int x = 0;
void testXDefaultVal() {
    assertEquals(0, x);
}
void testChangingXVal() {
    x = 1 + x;
    assertEquals(1, x);
}
```

Figure 1: Test code containing test dependence. If `testChangingXVal` executes before `testXDefaultVal`, then `testXDefaultVal` fails because `testChangingXVal` will have changed the value of `x` to be 1 by the time `testXDefaultVal` executes.

- False positive alarms, in which the test fails spuriously even when the software is correct. A failed test requires human attention, which is the most costly resource during software development.
- False negatives (missed alarms), in which the test passes even when a bug exists in the software (e.g., CLI bug [5]). A missed alarm delays discovery of a bug and increases the cost to fix it [4, 1, 15].

Figure 1 illustrates an example of test dependence. Test `testXDefaultVal` passes when it executes before `testChangingXVal`, but it fails when it executes after `testChangingXVal`.

Test dependence does exist in practice [17]. To cope with test dependence, tests could be required to run in a given order [10], but doing so prevents the use of test selection and prioritization. Another approach is to execute each test in a separate virtual machine [11], but doing so significantly increases test execution time. To the best of our knowledge, our approach is the first to automatically repair test dependence.

2. APPROACH

To repair test dependence is to refactor the tests so that each test in a suite yields the same result regardless of their execution order. Our proposed approach is to automatically repair the causes of test dependence during code compilation. More specifically, our approach automatically applies various refactorings to the code under test and test code as they are compiled.

There are three common root causes of test dependence [17]: at least 61% due to side-effecting access to shared global variables (e.g., static variables in Java), 10% due to side-effecting access to a database, and 4% due to side-effecting access to the file system. Since the most common root cause

```
static int x = 0;
void testXDefaultVal() {
    x = 0;
    assertEquals(0, x);
}
void testChangingXVal() {
    x = 0;
    x = 1 + x;
    assertEquals(1, x);
}
```

Figure 2: Repaired test code.

Table 1: Subject programs used in our evaluation. Column "LOC" represents the number of lines in the subject program's code under test (CUT) and human-written tests. Column "# Tests" shows the number of human-written unit tests and those generated by Randoop [12].

	LOC		# Tests		
Program	CUT	Tests	Human	Auto	Version
Crystal	4573	1302	78	3198	1.0.20111015
JFreechart	92255	49942	2234	2438	1.0.15
Joda-Time	27183	51492	3875	2234	b609d7d66d
Synoptic	5317	2758	118	2467	d5ea6fb3157e
XML Security	18255	3807	108	665	1.0.4

of test dependence is due to side-effecting access to shared global variables, we develop a prototype of our approach to specifically address this cause.

Our prototype works in two phases: (1) determine the initial value of every global variable (its value after all static initialization blocks have executed); and (2) for every test, reassign all occurrences of global variables whose value is potentially read during the test's execution with the value of the global variable obtained from Phase 1.

Static initialization blocks are called only when a class is initialized for the very first time and can be used to create and manipulate objects that can eventually be assigned to static variables. One challenge for Phase 1 is to record the creation and manipulation of objects inside static initialization blocks. Our prototype addresses this challenge by creating a mapping for each static variable to the assignments and manipulations performed on to the static variable.

Phase 2 of our prototype operates by statically analyzing the code under test and test code to determine which global variables a test may interact with and reassign such global variables. One challenge for this phase is to ensure that only one reassignment is created for global variables a test and methods invoked by this test may read. Our prototype addresses this challenge by creating a set for the global variables that may be read for each test and the methods invoked by the test, and finally reassigning these global variables in the beginning of the test.

For the example in Figure 1, when our prototype identifies that test `testXDefaultVal` reads `int x` and test `testChangingXVal` reads and writes `int x`, we repair the code such that both tests will reassign `int x` in the beginning of the test. An example of the repaired code can be seen in Figure 2.

3. RESULTS

Table 1 lists the subject programs used in our evaluation. The subject programs are known to contain dependent tests from our previous study [10]. It is notable that when these subject programs were chosen for our previous study, they were the first programs that we chose without knowledge of whether they actually contained dependent tests and yet all of them turned out to contain dependent tests. To measure the effectiveness of our prototype, we measure the number of dependent tests exposed by applying test prioritization [8, 9, 13, 14, 16] on the subject programs in Table 1 with and without first applying our prototype. We assess four different test prioritization algorithms from [6, Table 1].

Table 2: Number of unique dependent tests exposed by applying four test prioritization algorithms on the subject programs. Columns "Repaired" show the number of dependent tests exposed after applying our prototype.

	Number of dependent tests			
	Human tests		Auto tests	
Program	Original program	Repaired program	Original program	Repaired program
Crystal	5	2	51	43
JFreechart	3	3	5	5
Joda-Time	1	0	224	157
Synoptic	0	0	2	2
XML Security	4	4	78	77
Total	13	9	360	284

Out of the lower-bound number of dependent tests for these subject programs from a previous study [10], we find that our prototype eliminates 10.8% of human-written and 12.5% of automatically-generated dependent tests. Currently our prototype repairs only test dependence caused by shared global variables that are primitive or String typed. We believe that once we enhance the prototype to repair shared global variables of any type, then the prototype will be able to eliminate even more test dependence.

4. RELATED WORK

Our approach can automatically repair test dependence by refactoring the code under test and test code during code compilation. Previous research proposed other techniques to address test dependence.

The approach of Unit Test Virtualization [2] executes tests in multiple containers inside one JVM and Muşlu et al. [11] proposed executing tests in separate JVMs. In contrast to their approaches, our work can automatically repair the common causes of test dependence so that the repaired tests will yield consistent results when executed on a single, standard JVM. Other related work includes Electric Test [3] and PolDet [7]. Both of these approaches are capable of detecting test dependence but they do not repair test dependence.

5. FUTURE WORK

Currently, our prototype automatically fixes test dependence for only shared global variables that are of primitive or String types. In the future, we plan to continue development on this prototype to support variables of any type. Furthermore, the approach described in Section 2 can be applied to handle additional causes of test dependence such as access to a database or file system and concurrent programs. By covering additional causes of test dependence, we foresee the possibility for our tool to automatically repair all cases of test dependence.

6. REFERENCES

[1] W. Baziuk. BNR/NORTEL: Path to improve product quality, reliability, and customer satisfaction. In *Sixth International Symposium on Software Reliability Engineering*, Toulouse, France, October 24–27, 1995.

[2] J. Bell and G. Kaiser. Unit test virtualization with VMVM. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, pages 550–561, Hyderabad, India, June 4–6, 2014.

[3] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya. Efficient dependency detection for safe Java test acceleration. In *ESEC/FSE 2015: The 10th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 770–781, Bergamo, Italy, September 2–4, 2015.

[4] B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, 1976.

[5] An Apache CLI bug masked by dependent tests. https://issues.apache.org/jira/browse/CLI-26 https://issues.apache.org/jira/browse/CLI-186 https://issues.apache.org/jira/browse/CLI-187.

[6] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA 2000, Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pages 102–112, Portland, OR, USA, August 22–25, 2000.

[7] A. Gyori, A. Shi, F. Hariri, and D. Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 223–233, New York, NY, USA, 2015. ACM.

[8] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *ASE 2009: Proceedings of the 24th Annual International Conference on Automated Software Engineering*, pages 233–244, Auckland, NZ, November 18–20, 2009.

[9] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 119–129, Orlando, Florida, May 22–24, 2002.

[10] W. Lam, S. Zhang, and M. D. Ernst. When tests collide: Evaluating and coping with the impact of test dependence. Technical Report UW-CSE-15-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, March 2015.

[11] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), New Ideas Track*, pages 496–499, Szeged, Hungary, September 7–9, 2011.

[12] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 23–25, 2007.

[13] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[14] M. J. Rummel, G. M. Kapfhammer, and A. Thall. Towards the prioritization of regression test suites with data flow information. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 1499–1504, Santa Fe, NM, USA, March 14–17, 2005.

[15] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering*, pages 281–292, Denver, CO, November 17–20, 2003.

[16] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 97–106, Rome, Italy, July 22–24, 2002.

[17] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 385–396, San Jose, CA, USA, July 23–25, 2014.