Partial Ordering Unit Tests by Coverage Sets

Markus Gälli, Oscar Nierstrasz, Roel Wuyts

Institut für Informatik und Angewandte Mathematik
University of Bern, Switzerland

IAM-03-013

February 17, 2004

Abstract

A single software fault may cause several unit tests to break, if they cover the same methods. The coverage sets of unit tests may not just overlap, but include one another. This information could be of great use to developers who would like to focus on the most specific test that concerns a given fault. Unfortunately, existing unit testing tools neither gather nor exploit this information.

We have developed a simple approach that analyses a set of test suites, and infers the partial order corresponding to inclusion hierarchy of the coverage sets. When several unit tests in an inclusion chain break, we can guide the developer to the most specific test in the chain.

Our first experiments with three case studies suggest that unit tests for typical applications are, in fact, comparable to other unit tests, and can therefore be partially ordered. Furthermore, we found indications that this partial order is semantically meaningful, since faults that cause a unit test to break will, in nearly all cases cause less specific unit tests to break too.

CR Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging —*Testing tools*; D.3.3 [Programming Languages]: Language Constructs and Features —*Classes and objects*;

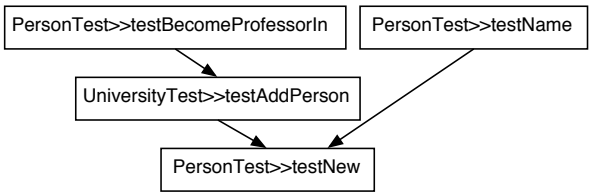

Figure 1: A Test hierarchy.

1 Introduction

Programmers are often confronted with the situation that a software change causes a number of unit tests to fail, but in fact only a single defect is responsible for multiple failures. An important question is therefore: Which test provides the best *focus* for the problem at hand, and will help the programmer to track down the defect most quickly?

In this paper we propose a partial order on unit tests – a unit test *covers* another unit test, if the coverage set of methods invoked by the first test is a superset of the coverage set of the second. We explore the hypothesis that this order can provide developers with the focus needed during debugging. By exposing this ordering, we can gain insight into the correspondence between unit tests and defects: if a number of related unit tests break, there is a good likelihood they are breaking because of a common defect; on the other hand, if unrelated unit tests break, we may suspect multiple defects. Consider, for example, the unit tests in Figure 1. We draw an arrow from one unit test to another if the first covers the second. The test method PersonTest≫testNew (*i.e.*, the method testNew of the class PersonTest) will invoke at run-time some set of methods of various classes. PersonTest≫testName will invoke those same methods, and more, so its coverage set includes that of PersonTest≫testNew. Note that we do *not* require that PersonTest≫testName invoke PersonTest≫testNew, or even that it tests remotely the same logical conditions; merely that it covers at least the same methods.

In this hierarchy, if all four unit tests break, we can infer that, with great likelihood, there is some single defect that is causing all the unit tests to break. Since PersonTest≫testNew is the "smallest" test, in the sense that it covers less methods, it provides us with better focus, and may help us find the defect more quickly. In any case, the fact that these unit tests are related will cause us to consider them as a group in the debugging process.

Unfortunately, existing unit testing tools do not order unit tests in terms of method coverage, and do not even gather this information.

Test prioritization [12] has been successfully used in the past to increase the likelihood that failures will occur early in test runs.

Here we investigate a different hypothesis. When multiple unit tests fail, the ones that cover one another fail due to the same defects. In this paper we provide initial evidence that:

- Most unit tests for a typical application are comparable by the *covers* relation, and can be partially ordered.
- When a unit test fails, a test that *covers* it typically fails too.

We have taken three case studies which had some unit tests, and analyzed the tests cases to infer the partial order. Then we introduced defects to test the hypothesis that when a unit

test fails, its covering unit tests will fail too. We validated this hypothesis in each of the three case studies.

In section 2 we present the experiments we carried out with three case studies. In section 3 we discuss our findings. In section 4 we give a brief overview of related work. In section 5 we conclude with a few remarks concerning future work.

2 Case studies

For our experiments, we took three separate case studies of small to medium size. The programs and unit tests of the case studies were created by three different developers who were not aware of our attempts to structure their tests while they were writing them. In each case we instrumented the subsystem being tested and generated a log of which methods were called by the unit tests. This information was then used to generate the partial order – the *coverage set* (the set of methods called) for each unit test was simply compared pairwise to that of every other unit test. If the coverage set of a test is a superset of a second test, we say that the first test *covers* the second. In a second phase, defects were introduced to validate that breaking of unit tests correlated to the partial order identified. In particular, if a unit test breaks, we expect that unit tests that cover it are likely to break as well.

The experiment was performed using VisualWorks Smalltalk. The code of interest was instrumented using AspectS [6]. Our tool removed unit tests that did not call any method of the instrumented subsystem but only called those of prerequisite subsystems. The resulting coverage sets were then compared pairwise, and used to compute a directed, acyclic graph representing the partial order. Unit tests with identical coverage sets were put in common nodes in the DAG, since they are equivalent in the partial order.

In a second phase, defects were introduced in the code to test the hypothesis that the failures would respect the ordering. We therefore

- iterated over all test cases that are covered by some other test case,
- determined which methods were invoked by that test, but not by any other test that it covers,
- mutated each method signature by deleting its method body,
- for each each mutation ran the unit tests and all its covering unit tests and collected the results.

2.1 MagicKeys

MagicKeys is a package that makes it easy to graphically view, change and export/import keyboard bindings in VisualWorks Smalltalk[1]. 16 unit tests existed for MagicKeys. 37% of the methods in the package were covered by the test cases. One test had an empty coverage set, which meant that it only checked methods of prerequisite packages. Indeed the comment of this test case read

> make sure that all the ctrl cmbinations are indeed in Textconstants. As the lookup in the dispatch-table depends on this (VisualWorks code, not ours), we

[1] http://homepages.ulb.ac.be/∼rowuyts/MagicKeys/index.html

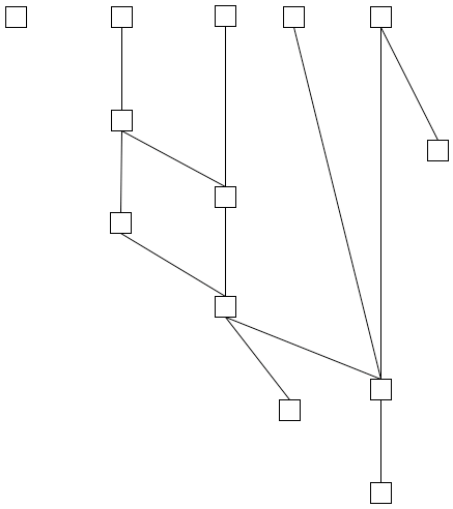

Figure 2: Magic Keys: Test hierarchy. Tests are displayed as squares. An edge from a lower square to an upper square means, that the lower unit test covers the upper one. Note that

assume they are. However, this tests this assumption (together with the naming convention).

Three unit tests were identified as equivalent. This equivalence is not surprising, given the names of them:

- MagicKeysTest≫testAltDispatchWriting
- MagicKeysTest≫testMetaDispatchWriting
- MagicKeysTest≫testShiftDispatchWriting

The resulting graph consisted of 12 nodes containing a single test case, and one containing the three equivalent unit tests. The computation of the inclusions took 19756 milliseconds to run whereas the uninstrumented unit test cases took 26 milliseconds to run.

Only one unit test was not comparable to any other test case (see Figure 3), and only 5 unit tests were not covered by any other unit test. (see Figure 2. The deepest level of a chain in the DAG was 6.

We mutated altogether 46 methods. 43 mutations also broke all covering unit tests. 23 of them yielded the same failing or erroneous test results in each covering test. So 20 mutations gave at least 2 different error or failure messages in their covering unit tests. 5 mutations of these gave both failures and errors in the covering unit tests. 3 mutations belonging to 3 different unit tests made the deepest unit test fail, but let some covering unit tests pass. This supports our hypothesis that most but not necessarily all unit tests break if some covered unit test fails.

2.2 VAN

VAN is a version analysis tool, which is being developed in our group. We examined its domain model and the associated unit tests.

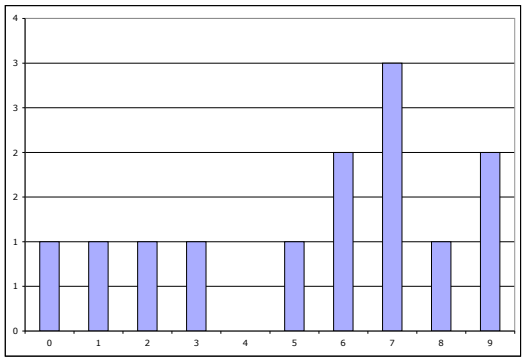

Figure 3: Comparable nodes of Magic Keys.

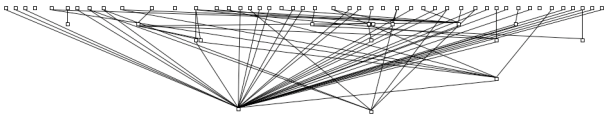

Figure 4: Van Test hierarchy.

The 67 unit tests covered 64% of the application. A total of 246 method signatures were covered. Running the uninstrumented unit tests alone took 100 seconds; running the instrumented unit tests and generating the partial order took 748 seconds.

Three pairs of equivalent unit tests were identified. One test case was identified as being empty so a total of 63 covered unit tests remained. The developer of VAN told us that one of these equivalent pairs consisted of an accidently duplicated test. In one of the remaining pairs one unit test was dependent on the other, but as they called the same methods in a recursive manner, our tool could not differentiate them.

16 of the unit tests where covered in at least one other unit test. This means that for roughly every fourth unit test the chance is very high, that if it fails at least one other unit test will fail also.

The deepest level of a chain in the DAG was 5. We asked the developer of VAN, where he would expect some overlapping before we started the experiment. He predicted 2 places which were included in the 13 found.

The resulting graph of TestComposer contained the following coverage between unit tests, where their composition property could have even been guessed by their method names:

- CategoryHistoryGroupTest≫testENM
  (OperatorTest≫testENMOperator)
  We introduced a bug in the ENMOperator class and both tests naturally failed.

- LoaderTest≫testConvertXMIToCDIF
  (LoaderTest≫testLoadXMI)

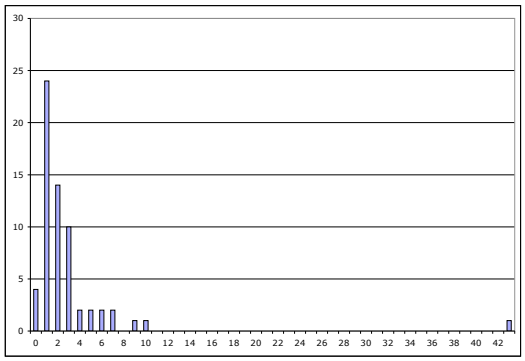

Figure 5: Comparable nodes of VAN.

- SystemHistoryTest≫testAddVersionNamedCollection
  (SystemHistoryTest≫testAddVersionNamed)
- SystemHistoryTest≫testSelectClassHistoriesWithLifeSpan
  (SystemHistoryTest≫testSelectClassHistories)

We mutated altogether 251 methods. 160 mutations did not let any unit test pass. 130 of them yielded the same failing or erroneous test results in each covering test. So 121 mutations gave at least 2 different error or failure messages in their covering tests. Only 23 out of 251 hard method deleting mutations broke the deepest unit test in the coverage chain without breaking all the covering unit tests. To do the mutation experiment took about 120 minutes.

2.3 CodeCrawler

CodeCrawler [7] is a language independent reverse engineering tool which combines metrics and software visualization and is also being developed in our group. We examined all its 79 unit tests. The 79 unit tests covered 24% of our instrumented methods and took 51 seconds to run uninstrumented. The sorting and running of the instrumented unit tests took 261 seconds. Four of the 79 unit tests only checked methods of prerequisite systems and our tool therefore declared them as being empty. 28 of the remaining 75 unit tests could be put into 10 equivalence relations. Here again we have an example of equivalent unit tests whose names suggest their similarity.

- CCNodeTest≫testRemovalOfEdgeRemovesChild
- CCNodeTest≫testRemovalOfEdgeRemovesParent
- CCNodeTest≫testRemovalOfSoleEdgeRemovesChildOrParent

So 57 unit tests were sorted and only 8 were incomparable to any other node. The deepest level of a chain in the DAG was 6. 36 unit tests were isolated, 25 unit tests were covered by at least one other test. One unit test was covered by many unit tests, because it was mainly using some setup.

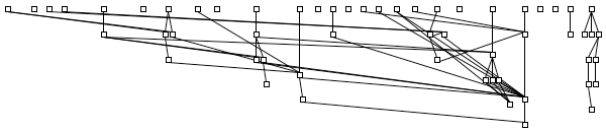

Figure 6: Code Crawler Test Hierarchy (displayed using Code Crawler).

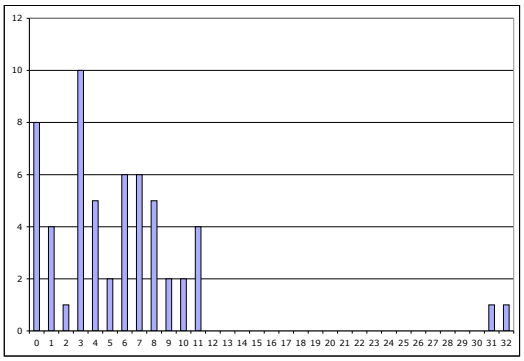

Figure 7: Comparable nodes of CodeCrawler.

Unfortunately, we were unable to run the full experiment, which appeared to loop endlessly due to some of the defects we had introduced. In order to get some results, we chose to only mutate a 10th of the covered methods of each test case.

We mutated altogether 41 methods. 27 mutations yielded different results for the chain of dependent unit tests and 14 the same. In 11 out of the 27 mutations no unit test passed. Only 3 mutations broke the inner test but some covering unit test passed. To do the mutation experiment took about 45 minutes.

3 Discussion

The experiments described above are rather simple, but they are also remarkable for the consistency of their results. In each case, a significant majority of the test cases was comparable to other unit tests, using the rather stringent criterion of inclusion of the sets of methods called. We fully expect to obtain similar results for other applications with good test coverage.

Furthermore, each of the case studies consistently showed that if a defect will cause a particular unit test to break, then, with high probability, unit tests that precede it in the partial order will also break. The partial order over tests is therefore, not just accidental, but exposes semantic relationships between the tests.

These relationships should, in principle, be of aid to developers who must track down defects in the software when changes are introduced. Whenever a change causes multiple unit tests to break, the partial order over the unit test set can be used to tell the developer which

are the most *specific* unit tests that have broken (*i.e.*, those with the smallest coverage sets). These unit tests will likely give the developer the best focus when debugging. Less specific unit tests are probably breaking for the same reason, and may only introduce more noise into the debugging process.

We must be careful, though, not to claim too much. We have only validated the hypothesis that a partial order over a typical test set exists, and that this order is meaningful. Let us now consider the limitations of what we know.

The defects we have introduced are quite radical – we have eliminated entire method bodies of methods being called in known tests. Will other, more realistic kinds of defects cause chains of unit tests breaking in ways that respect the partial order?

We compare a unit test to another using the set of *all* instrumented methods called, without distinguishing methods called directly or indirectly. Would we obtain similar results considering only direct method calls? AspectS provides us with enough detailed information to allow us to perform other experiments that make use of more fine-grained information.

Similarly, we make no distinction between methods that are being called incidentally, and those methods that are actually being tested. In fact, there is no way to generally know which method is being tested without asking the test developer. Certain methods will be called to set up the test objects, others to obtain certain side effects, yet more to sample the state of the test objects, and others to clean up. It is hopeless (if not meaningless) to detect automatically which method is being "tested" most.

We have not yet made a distinction between failures, *i.e.*, unit tests that break because of a failed assertion, and errors, *i.e.*,unit tests that fail because a run-time exception occurs. We expect that failures will mostly respect the partial order, since they indicate that something that is intentionally being tested has broken. It seems naturally that a less specific test will likely break as well. But an error is more accidental in nature. Since each unit test case is different (we observed very few instances of unit tests actually calling other unit tests), there is no reason to expect errors to respect the partial order. At this time, however, we have no evidence that there is a difference between failures and errors.

There is an implicit assumption in our motivation for this work that more specific unit tests in the partial order are actually "smaller" in some sense than less specific unit tests, and that they will actually provide better focus in debugging. Both these hypotheses are open at this time. Nevertheless, even if we obtain evidence to the contrary, we expect that developers will still be able to benefit from the knowledge that certain tests are semantically related, even if the partial order does not necessarily correspond to a natural order in which to examine the tests.

Naturally there is a cost to identifying the partial order. Potentially this order will change every time the system is modified, or a test case is added or changed, so the order will have to be recalculated. The cost of instrumenting the tests and calculating the partial order must be small enough that it does not pose an unacceptable burden. The current approach of pairwise comparisons is admittedly naive and inefficient, but we can easily imagine more efficient ways of topologically sorting unit tests, and incrementally updating the order when changes occur.

4 Related Work

Test-driven development [1] is a technique in which testing and development occur in parallel, thereby providing developers with constant feedback. Saff and Ernst [13] demonstrate that

continuous testing speeds up development. Parrish *et al.* [9] define a process for test-driven development that starts with fine-grained tests and proceeds to more coarse-grained tests.

> Once a set of test cases is identified an attempt is made to order the test case runs in a way that maximizes early testing. This means that defects are potentially revealed in the context of as few methods as possible, making those defects easier to localize.

In their approach, tests are written with a particular order in mind. In our approach, on the other hand, we investigate *a posteriori* orderings of existing tests. In both cases, the ultimate goal is to improve the quality the feedback loop.

Debugging with tests is analogous to grading students with exams. If individual exam questions are too broad, one can only detect that a student is deficient in some area, but not be able to tell specifically what knowledge is missing. In oral exams, teachers may move from broader to more specific questions to tell exactly where a student is strong or weak. In computer aided teaching [4] the term "granularity" was introduced with this sense of more specific and less specific questions. The partial order we impose on test sets are intended to capture a similar notion of granularity for software tests, though the resulting "granularity hierarchies" [8] are technical quite different from the partial orders we obtain.

Rothermel *et al.* [10] introduce the term "granularity" for software testing, but they focus on cost-effectiveness of test suites rather than on debugging processes.

Selective regression testing is concerning with determining an optimal set of tests to run after a software change is made [11] [2]. Although there are some similarities with the work initiated in this paper, the emphasis is quite different. Instead of selecting which tests to run, we analyse the set of tests that have *failed*, and suggest which of these should be examined first.

Test case prioritizing [12] sorts test cases to some specific criteria. The criterion which most closely matched our approach was *total function coverage* [5]. Here a program is instrumented and, for any test case, the number of functions in that program that were exercised by that test case is determined. The test cases are then prioritized according to the total number of functions they cover by sorting them in order of total function coverage achieved.

Zeller, *et al.* [3][14] use delta debugging on simplifying test case input, reducing relevant execution states and finding failure-inducing changes. We focus on reducing failing tests from a set of semantically different tests to the most concise but still failing tests, their techniques could pay off even more using this smaller tests.

5 Conclusion and Future work

We have proposed a simple approach to partially order unit tests in terms of the sets of methods they invoke. Initial experiments with three case studies reveals that this simple technique exposes a large number of ordering relationships between otherwise independent tests. Furthermore, the partial order seems to correspond, in most cases, to a semantic relationship in which less specific unit tests tend to fail if more specific unit tests also fail.

The reported experiments are only a first step. We plan to explore much larger case studies, and see if these results scale up. The correspondence between the partial order and failure dependency between unit tests needs to be tested with other kinds of defects. We plan to artificially introduce more fine-grained defects and check their impact on test failures, and

we also plan to analyse historical test failure results for their correspondence with the partial order.

So far our experiments have been limited to Smalltalk, but it should be easy to extend the approach to other languages, like Java.

In the long term, we are interested in exploring the impact of ordering and structuring tests on the development process. The partial order that is detected automatically may not only help to guide developers in the debugging process, but it may provide hints to how tests can be better structured, refactored and composed.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004) and "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

References

[1] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley, 2003.

[2] John Bible, Gregg Rothermel, and David Rosenblum. A comparative study of coarse- and fine-grained safe regression test selection. *ACM TOSEM*, 10(2):149–183, April 2001.

[3] Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. In *Proceedings of the Fourth International Workshop on Automated Debugging*, August 2000.

[4] Jason A. Collins, Jim E. Greer, and Sherman X. Huang. Adaptive assessment using granularity hierarchies and bayesian nets. In *Proceedings of the Third International Conference on Intelligent Tutoring Systems*, pages 569–577, 1996.

[5] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, 2000.

[6] Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays 2002*, pages 216–232, Erfurt, 2003. Springer.

[7] Michele Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.

[8] Gordon McCalla, Jim Greer, Bryce Barrie, and Paul Pospisel. Granularity hierarchies. In Fritz Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 363–375. Pergamon Press, Oxford, 1992.

[9] Allen Parrish, Joel Jones, and Brandon Dixon. Extreme unit testing: Ordering test cases to maximize early testing. In Michele Marchesi, Giancarlo Succi, Don Wells, and Laurie

Williams, editors, *Extreme Programming Perspectives*, pages 123–140. Addison-Wesley, 2002.

[10] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings ICSE-24*, pages 230–240, May 2002.

[11] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[12] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings ICSM 1999*, pages 179–188, September 1999.

[13] David Saff and Michael D. Ernst. Can continuous testing speed software development? In *Fourteenth International Symposium on Software Reliability Engineering ISSRE 2003*. IEEE, November 2003.

[14] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, SE-28(2):183–200, February 2002.