Ordering Broken Unit Tests for Focused Debugging

Markus Gälli, Michele Lanza, Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
{gaelli,lanza,oscar}@iam.unibe.ch

Roel Wuyts
Lab for Software Composition and Decomposition
Université Libre de Bruxelles
roel.wuyts@ulb.ac.be

Abstract

Current unit test frameworks present broken unit tests in an arbitrary order, but developers want to focus on the most specific ones first. We have therefore inferred a partial order of unit tests corresponding to a coverage hierarchy of their sets of covered method signatures: When several unit tests in this coverage hierarchy break, we can guide the developer to the test calling the smallest number of methods. Our experiments with four case studies indicate that this partial order is semantically meaningful, since faults that cause a unit test to break generally cause less specific unit tests to break as well.

Keywords: Unit testing, debugging

1. Introduction

Unit testing has become increasingly popular in recent years, partly due to the interest in agile development methods. [1]

Since one fault can cause several unit tests to break, the developers do not know which of the broken unit tests gives them the most specific debugging context and should be examined first.

We propose a partial order of unit tests by means of *coverage sets* — a unit test A *covers* a unit test B, if the set of method signatures invoked by A is a superset of the set of method signatures invoked by B.

We explore the hypothesis that this order can provide developers with the focus needed during debugging phases. By exposing this order, we gain insight into the correspondence between unit tests and defects: if a number of related unit tests break, there is a good chance that they are breaking because of a common defect; on the other hand, if unrelated unit tests break, we may suspect multiple defects. The key to make the unit test suite run again is to identify the central unit tests that failed and thus caused a "failure avalanche" effect on many other tests in the suite.

The results of four case studies are promising: 85% to 95% of the unit tests were comparable to other test cases by means of their *coverage sets* – they either covered other unit tests or were covered by them. Moreover, using method mutations to artificially introduce errors in a test case, we found that in the majority of cases the error propagated to all test cases covering it.

Structure of the article. In Section 2 we describe the problem of implicit dependencies between unit tests. In Section 3 we then describe our solution to this problem. In Section 4 we present the experiments we carried out with four case studies. In Section 5 we discuss our findings. In Section 6 we give a brief overview of related work. In Section 7 we conclude and present a few remarks concerning future work.

2. Implicit dependencies between unit tests

Example. Assume we have the following four unit tests for a simplified university administration system:

- PersonTest≫testBecomeProfessorIn tests if some person, after having been added as a professor also has this role.
- UniversityTest≫testAddPerson tests if the university knows a person after the person has been added to it.
- PersonTest≫testNew tests if the roles of a person are defined.
- PersonTest≫testName tests if the name of a person was assigned correctly.

For a detailed look at the run-time behavior of the test cases see Figure 1 and Figure 2.

Furthermore assume that the implementation of Person class≫new is broken, so that no roles are initialized and the role variable in Person is undefined. When we run the four tests, two of them will fail:

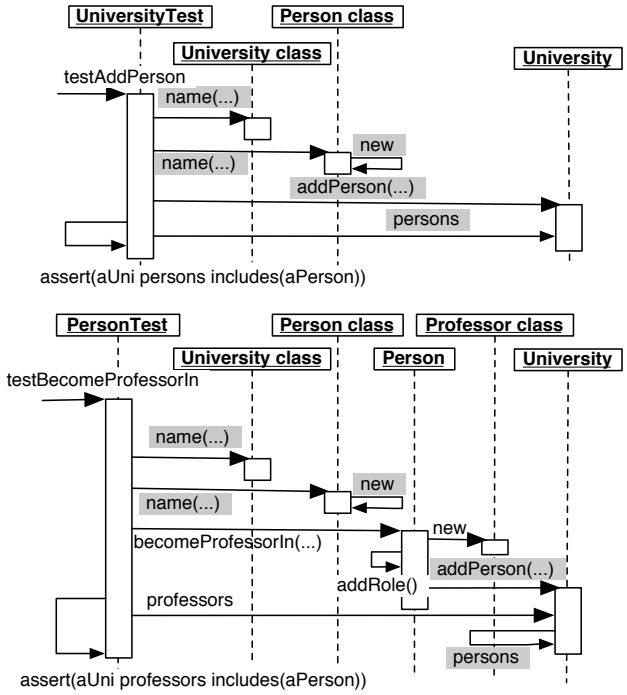

Figure 1. The test for #becomeProfessorIn: covers the test for #addPerson:. Intersecting signatures are displayed gray.

1. The test PersonTest≫testBecomeProfessorIn (see Figure 1) yields a null pointer exception: *Undefined object does not understand: add:* occurring in Person≫addRole:.

2. In test PersonTest≫testNew (see Figure 2) the assertion person roles notNil fails, pointing directly to the problem at hand.

As the latter failing test case provides the developer directly with the information needed to fix the error, the latter one should be presented first. We therefore order the unit tests according to their sets of covered methods. All the methods which are called in PersonTest≫testNew are also called in UniversityTest≫testAddPerson (see Figure 1 and Figure 2). Again all methods sent by UniversityTest≫testAddPerson are themselves included in the set of methods sent by PersonTest≫testBecomeProfessorIn (see Figure 1). Note that PersonTest≫testName is neither covered by any other test nor covering one.

Consider the unit tests in Figure 3. We draw an arrow from one unit test to another if the first *covers* the second, as defined in Section 1. The test method PersonTest≫testNew (*i.e.,* the method testNew of the class PersonTest) will invoke at run-time a set of methods of various classes. PersonTest≫testBecomeProfessorIn will invoke at least those same methods, so its coverage set includes that of PersonTest≫testNew. Note that we do *not* require that PersonTest≫testBecomeProfessorIn invoke PersonTest≫testNew, or even that it test remotely the same logical conditions; merely that at least the same methods be executed during the test run.

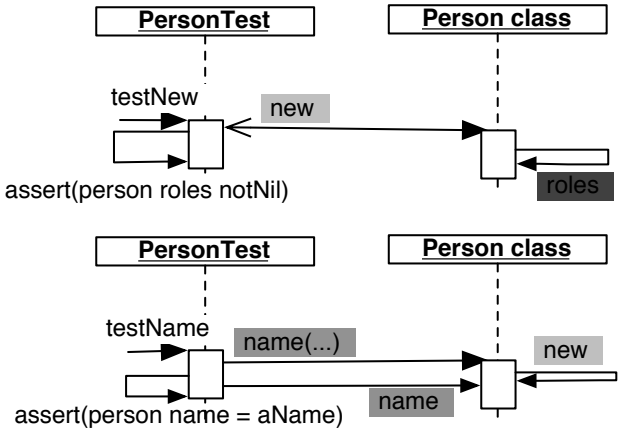

Figure 2. Two small unit tests, which do not cover each other.

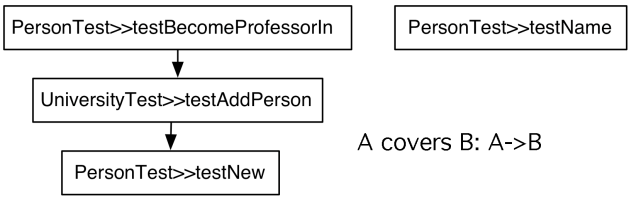

Figure 3. A sample test hierarchy based on coverage sets.

Unfortunately, existing unit testing tools and frameworks do not order unit tests in terms of method coverage, and do not even collect this information. In this paper we investigate the following hypothesis: When multiple unit tests fail, the ones that cover one another fail due to the same defects. We provide initial evidence that:

- Most unit tests of a typical application are comparable by the *covers* relation, and can thus be partially ordered.

- When a unit test fails, another test that *covers* it typically fails too.

If unit tests break in the same coverage chain of our coverage hierarchy, we can infer that there is a single defect that is causing all unit tests to break. Since PersonTest≫testNew is the "smallest" test (in the sense that it covers the least methods), it provides us with better focus, and helps us find the defect more quickly. In any case, the fact that these unit tests are related makes us consider them as a group in the debugging process.

3. Ordering broken unit tests

In this section we explain our approach of ordering in detail, and discuss an implementation in a Smalltalk environment. The problem we tackle is to infer coverage hierarchies, given a set of unit tests. We therefore need to generate traces and then order them.

3.1. Approach

To order the tests we used dynamic analysis because we

- have runnable test cases
- could apply it to both dynamically and statically typed languages
- and are only interested in the actual paths taken of our unit tests

The examined unit tests are all written in SUnit, the Smalltalk version of the XUnit series of unit test frameworks that exist for many languages. Our approach is structured as follows:

1. We create an instance of a *test sorter*, into which we will store the partially ordered test cases.
2. We iterate over all unit tests of a given application. We instrument all methods of the application so that we can obtain trace information on the messages being sent. The exact instrumentation mechanism to obtain the information depends on the implementation language. We used the concept of *method-wrappers* ([4]), where the methods looked up in the method dictionary are replaced by wrapped versions, which can trigger some actions before or after a method is executed. Here the method wrapper simply stores if its wrapped method was executed.
3. We then
   (a) execute each unit test, in our case via the XUnit-API,
   (b) obtain the set of method signatures which were called by the test, in our case by iterating over all wrapped methods and checking if they have been executed,
   (c) check if this set is empty, which for example could be due to the fact that the test only called methods of prerequisite packages,
   (d) if the set is not empty, we create a new instance of a *covered test case*, where we store this set of method signatures together with the test,
   (e) add this *covered test case* to the *test sorter*,
   (f) reset the method wrappers, so that they are ready to store if the next unit test executes them.
4. Some of the *covered test cases* are equivalent to others as their sets of covered method signatures are equal. To obtain a partial order we have to subsume this equivalent *covered test cases* under one node, that we call an *equivalent test case*. For all equivalent *covered test cases* we create an instance of an *equivalent test case*, store the set of method signatures and the names of the equivalent test cases in it, store it in the *test sorter* and then remove the equivalent *covered test cases* out of the *test sorter*. Note that both *covered test cases* and *equivalent test cases* are *test nodes*, a superclass where we store the shared behavior of this two.
5. We then order the resulting *test nodes* stored in our *test sorter* using the following relationship: A *test node* A is smaller than a *test node* B if the set of method signatures of A is included in the set of method signatures of B. We therefore pairwise compare the remaining *test nodes* and thus build a partial order. We store both the covering and the being covered relationship in variables of the *test node*.
6. Finally we compute the transitive reduction of this lattice, thus eliminating all redundant covering relations between the test nodes.
7. Finally we obtain an instance of a *test sorter* that we can ask which of some given tests we should attack first. Note that we did the case studies with non breaking unit tests. In the real world scenario with broken unit tests, we could either use a *test sorter*, which was initialized with the tests while they were non breaking, or reinitialize it with only the broken unit tests.

3.2. Implementation

In order to perform experiments to validate our claim, we implemented our approach in VisualWorks Smalltalk[1]. We

[1] See www.cincomsmalltalk.com for more information.

chose to do the implementation in VisualWorks Smalltalk because

- tools to wrap methods and assess coverage are freely available,
- we have numerous case studies available,
- we can build on the frrely available tool CodeCrawler [12] to visualize the information we obtained.

We obtain the trace information by using AspectS [9], a flexible tool which builds upon John Brant's *MethodWrappers* [4]. Though AspectS obtains the traces in the same way as method-wrappers described before, we used AspectS because it lets us obtain more detailed information about the current state of the stack, when a method is entered. In Java we could use *AspectJ* [11].

4. Case studies

We performed our experiments on the following four systems, which were created by four different developers, who were unaware of our attempts to structure their tests while they were writing them.

1. *MagicKeys*[2], an application that makes it easy to graphically view, change and export/import keyboard bindings in VisualWorks Smalltalk.
2. *Van* (Gîrba *et al.* [8]), a version analysis tool built on top of the Moose Reengineering Environment [6].
3. *SmallWiki* (Renggli [15]), a collaborative content management tool.
4. *CodeCrawler* (Lanza [12]), a language independent reverse engineering tool which combines metrics and software visualization.

4.1. Setup of the experiments

In a first phase, we ordered the unit tests for each case study as described in Section 3 and measured if a relevant portion of them were comparable by our *coverage* criterium.

In a second phase, we introduced defects into the methods to validate that if a unit test breaks, its covering unit tests are likely to break as well. We therefore

1. iterated over all test cases of the case study that were covered by at least one other test case,
2. determined which methods were invoked by each of those tests, but not by any other test it is covered by,
3. mutated the methods according to some mutation strategy,
4. and, for each each mutation, executed the unit tests and all its covering unit tests and collected the results.

We used the following mutation strategies:

1. *full body deletion*, *i.e.*, we removed the complete method body.
2. code mutations of *JesTer* [13]: JesTer is a mutation testing extension to test JUnit tests by finding code that is not covered by tests. JesTer makes some change to the code, runs the tests, and if the tests pass, JesTer reports what it changed. We applied the same mutations as JesTer, which are
   (a) change all occurrences of the number 0 to the number 1
   (b) flip true to false and vice versa
   (c) change the conditions of ifTrue statements to true and the conditions of ifFalse statements to false.

4.2. Results

The case studies are presented at more detail in Table 1 and Table 2.

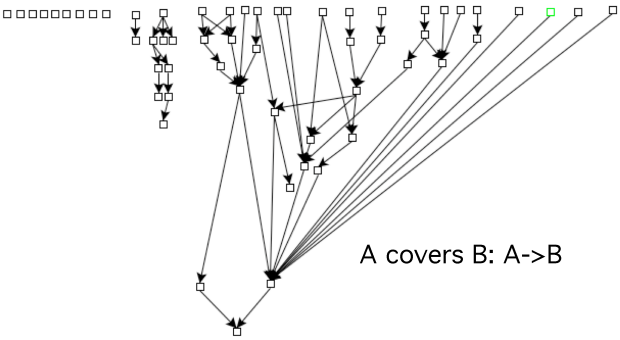

Figure 4. The coverage hierarchy of the Code Crawler tests visualized with Code Crawler.

As we see in Table 1 our experiment was performed with applications which had 1600 to 5600 lines of code. The ratio of LOC(Tests) to LOC reached from 13 % to 56%. The maximum test coverage was 64%.

In Figure 4 an arrow from the top to bottom denotes that the *test node* at the top *covers* the *test node* at the bottom. We see a typical coverage hierarchy obtained in the first part of our experiment: Most of the unit tests either covered or

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

System	LOC	LOC (Tests)	Coverage	#Unit Tests	Equivalent tests	Tests covered by Tests
Magic Keys	1683	224	37%	15	20%	53.3%
Van	3014	716	64%	67	9%	24.2%
CodeCrawler	4535	1071	24%	79	37.3%	40%
SmallWiki	5660	3096	64%	110	29.8%	47.4%

Table 1. The resulting coverage of unit tests in our case studies.

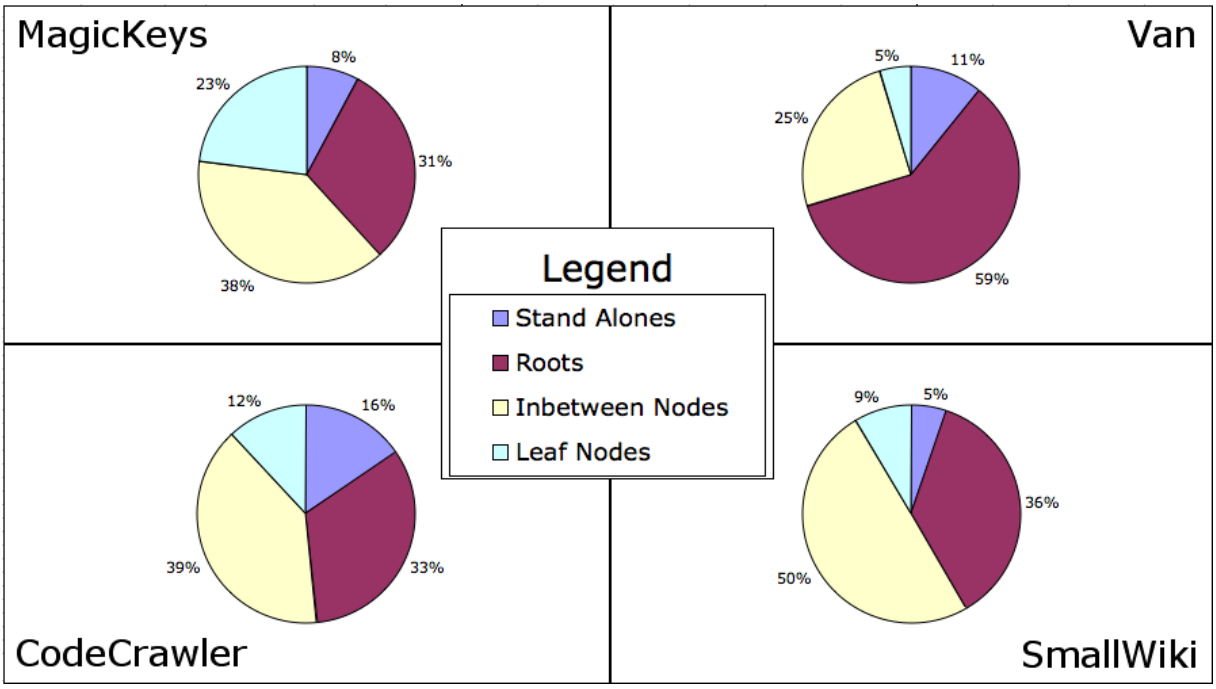

Figure 5. The distribution of comparable test nodes in our four case studies.

were covered by some other unit test and only 5% to 16% of them were stand alones (Figure 5).

A considerable percentage of unit tests (9% to 37%, see Table 1) called the same set of method signatures as at least one other test. 25% to 53% of the unit tests were covered by at least one other unit test. This means that for roughly every third test of our case studies, the probability is high, that if the test fails, it will not fail alone.

We carried out the second phase of our experiment, the automatic method mutation, in all case studies except CodeCrawler. As many mutations in CodeCrawler resulted in endless loops we did not have time to complete it. We merely did the full deletion mutation on every 10th method and omitted the JesTer mutations. The results are displayed in Table 2: 92% to 99.5% of the full deletion mutations of a method broke the smallest test calling this method and all its covering tests, as did 59% to 100% of the JesTer mutations. Note that the number of mutated methods is larger than the number of methods, as the same method could be mutated in the context of different tests.

Let us have a detailed look at the effects of a full method deletion on the coverage hierarchy of the Magic Key tests in Figure 6. We are mutating a method which is called from the test MagicKeysTest≫testMasks, thus from all of its covering tests. Here we picked a rare example, where not all of the covering tests are failing. Both MagicKeysTest≫testRegularCharCreating and the node including the equivalent test cases MagicKeysTest≫testMetaDispatchWriting, testAltDispatchWriting and testShiftDispatchWriting do not fail because of the deleted method. On the other hand the two tests MagicKeysTest≫testSpecialConstantKeyCreating and MagicKeysTest≫testKeyCopying also fail, though they do not cover the test MagicKeysTest≫testMasks, they merely have a non-empty intersection set with it, including the mutated method. Also note, that MagicKeyTest≫testKeyCopying, which is a standalone test, has the lowest number of method signatures called, and not

System	#Methods	Strategy	#Methods mutated	Errors propagating to all covering tests
Magic Keys	277	Full Deletion	46	93.5%
		JesTer	17	58.8%
VAN		Full Deletion	357	97.8%
		JesTer	59	100%
CodeCrawler	1104	Full Deletion	41	92.7%
SmallWiki	1565	Full Deletion	2415	99.5%
		JesTer	318	100%

Table 2. Results of our automatic mutation experiments.

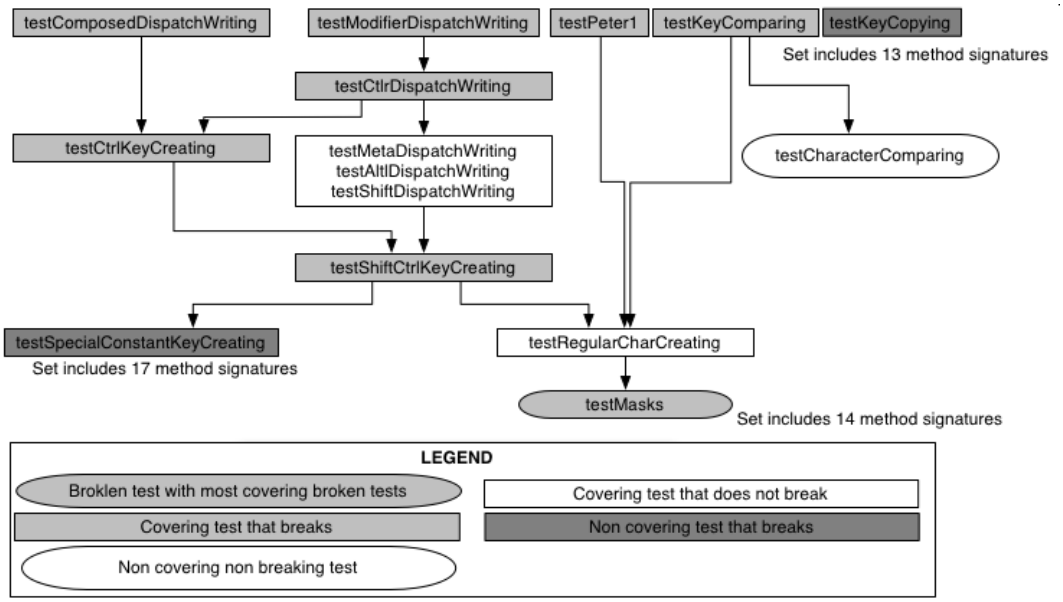

Figure 6. An avalanche effect in the coverage hierarchy of Magic Keys. One manually introduced bug causes 10 test cases to fail.

MagicKeysTest≫testMasks.

5. Discussion

The experiments we performed 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 called methods. Furthermore, each case study consistently showed that if a defect causes a particular unit test to break, unit tests that precede it in the partial order also tend to break. The partial order over tests is therefore not accidental, but exposes implicit dependency relationships between the tests.

5.1. Semantic ordering of tests

In this paper we focused on bug tracking via partial ordering of unit tests. Providing the order of unit tests could also help the developer to comprehend the structure of the unit tests and the structure of the underlying system. It can reassure the developer in his or her perceived layering of the system if the order of the test cases reflects this layering.

The method names of the example in Table 3 indicate a parallel structure of the tests, while the method names in the list below suggest a hierarchical one:

- LoaderTest≫testConvertXMIToCDIF (LoaderTest≫testLoadXMI)
- SystemHistoryTest≫testAddVersionNamedCollection (SystemHistoryTest≫testAddVersionNamed)

System	Signature of test case
Magic Keys	MagicKeysTest≫testAltDispatchWriting
Magic Keys	MagicKeysTest≫testMetaDispatchWriting
Magic Keys	MagicKeysTest≫testShiftDispatchWriting
CodeCrawler	CCNodeTest≫testRemovalOfEdgeRemovesChild
CodeCrawler	CCNodeTest≫testRemovalOfEdgeRemovesParent
CodeCrawler	CCNodeTest≫testRemovalOfSoleEdgeRemovesChildOrParent

Table 3. Examples for equivalent test cases.

- SystemHistoryTest≫ testSelectClassHistoriesWithLifeSpan (SystemHistoryTest≫testSelectClassHistories)

5.2. Limitations

The lightweight nature of our approach has some drawbacks and limitations:

- One unexpected result was that if the JesTer mutations were applicable to some unit tests, in 100% of the cases a broken inner test case meant that all its covering tests were broken. Thus our first assumption that the more specific JesTer mutations would let more covering test cases survive, seems to be incorrect: The JesTer method tweaks are even more fatal to the majority of covering tests than full body deletions. We plan to use more realistic mutations and manual introduction of errors in future experiments to overcome this problem.
- Parallel tests seem to cover each other even if they differ only by one method signature. Sorting these basically equal unit tests does not add an advantage as any exception of them will be as telling as the other.
- So far we have limited our case studies to Smalltalk programs. Perhaps style and conventions used in Smalltalk produce results which differ in other object-oriented languages.
- The developers of the case studies are all members of our research group thus also working in academia: We plan to make case studies with programs developed in industrial settings.
- We have not yet measured the implications of real bugs. How many unit tests break because of just one real bug and not because of one artificial mutation?
- We did not make any distinction between failures and errors when we were evaluating the chain of failed tests caused by one mutation.

6. Related Work

Unit testing has become a major issue in software development during the last decade: Test-driven development (TDD) [1] is a technique in which testing and development occur in parallel, thereby providing developers with constant feedback. The most popular unit testing framework used in TDD named XUnit [2] does not currently prioritize failed unit tests.

Parrish *et al.* [14] define a process for test-driven development that starts with fine-grained tests and proceeds to more coarse-grained tests. They state that "*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 beforehand with a particular order in mind, while in our approach we investigate *a posteriori* orderings of existing tests.

Rothermel *et al.* [16] 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 concerned with determining an optimal set of tests to run after a software change is made [17] [3]. Although there are some similarities with the work described 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 prioritization [18] has been successfully used in the past to increase the likelihood that failures will occur early in test runs.The tests are prioritized using different criteria, the criterion which most closely matched our approach was *total function coverage* [7]. 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, starting with the highest.

Wong *et al.* [19] compare different selection strategies for regression testing and propose a hybrid approach to se-

lect a representative subset of tests combining modification based selection, minimization and prioritization. Again, they emphasize on which tests should be run and not on how failing tests should be ordered. Modification based selection is their key to minimize the number of tests to run, thus they are relying on having prior versions of the tested program whereas our approach can in principle be used without having prior versions, as we could also order the tests using only the coverage of the failed tests.

Zeller *et al.* [20] [5] use delta debugging to simplify 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. Thus the technique of Zeller *et al.* could pay off more using this smaller tests as initial input.

7. Conclusion and future work

We have proposed a lightweight approach to partially order unit tests in terms of the sets of methods they invoke. Initial experiments with four case studies reveal that this technique exposes important implicit ordering relationships between otherwise independent tests. Furthermore, our experiments show that the partial order corresponds 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 analyze historical test failure results for their correspondence with the partial order. Moreover, so far our experiments have been limited to Smalltalk; we plan to extend the approach to other languages such as Java.

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

We believe that the research presented in this paper is but a first step in using coverage information to get more information from failing tests. The next section describes some ideas we got for future experiments while validating the claims from this paper.

7.1. Defect tracking

The coverage relationships could aid developers to 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 identify the most *specific* unit tests that have broken (*i.e.*, those with the smallest coverage sets). Identifying these unit tests gives 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.

Currently we order the unit tests based the sets of covered method signatures they produce while they run without failure. Thus our sets might be unnecessarily big and we need prior versions of the system under test. We want to compare this approach with one where only the coverage of the *failed* test cases is taken into account.

7.2. Other lightweight metrics for sorting tests: Testing time and size of coverage sets

Having given evidence that the situation of depending unit tests occurs on a regular basis, we can seek more lightweight variants to our approach: Can we for example also use testing time as an equivalent mechanism to order the tests? Unit tests which are covered by other unit tests might not be faster executed than the covering ones:

- The methods of the inner test might occur in some loop while the outer ones are only executed once.
- In languages with garbage collectors the testing time can vary.

Using the size of the sets of covered method signatures is more practical.To be sure to get the most *specific* unit test first, it is sufficient to sort the test cases by the size of their covered sets, starting with the lowest. There the program has to be instrumented and the tests sorted also. But our admittedly simple way of pairwise comparing the test cases can be omitted which could lead to faster results. Using this metric alone would have made it difficult on the other hand, to show the semantic dependencies of unit tests we have presented in this paper.

7.3. Using pre- and postcondition but keeping the scenarios

Looking back to our small example from Figure 2, one could gain the same prioritizing effect with a little code refactoring: The assertion in PersonTest≫testNew could be used in the program as a postcondition and should be moved into Person class≫new. Then *all* covering tests would also immediately fail at this very same position giving the developer the most specific information about the problem at hand. This refactoring would even make the PersonTest≫testNew test superflous, as long as one knows that Person class≫new is executed by some test case.

The test code should be reduced to compose scenarios, execute methods on this scenarios, and possibly deliver the

result of this executions for further scenario building. Identifying the high level scenarios necessary to run all sub scenarios would lead to a massive reduction of testing time, massive reuse of assertions also in unexpected scenarios, and would make our post mortem sorting approach unnecessary: The most specific assertions will always fail first, directing the developer immediately to the problem at hand. In the four case studies we analyze in this paper, none of the software developers wrote any pre- or post-condition or invariant, thus relying solely on the assertions in their unit tests. This is a common behavior of Smalltalk developers today: The open source Smalltalk environment *Squeak* [3] [10] in the version from February 2004 includes 1024 unit tests but only 23 pre- or post-conditions.

Methods like XP suggest frequent testing and developing in small increments, so that developers can identify their latest changed code as a good starting point to know where an error is. But to know why the error occurred, they want to go to the most detailed test with the most focused assertion. Our experiments and experience show, that one failed unit test comes seldom alone, so unless they start putting assertions in the code, they still need to find the most specific of them.

This approach of combining design by contract and classical unit testing is facing several problems:

- It requires manual refactoring of the test cases and the program.
- Unit tests without assertions seem like a self contradiction and pose a mental barrier to developers.
- Missing explicit relationships between between unit tests and methods under test make it hard to tell what scenarios are covering a method containing a postcondition.
- Moreover missing integration of coverage browsers in current IDEs makes it hard to tell if some test scenario even executes a method containing a postcondition.

Acknowledgments

We thank Orla Greevy and Tudor Gîrba for helpful comments. 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).

[3] See http://www.squeak.org for more information.

References

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

[2] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[3] J. Bible, G. Rothermel, and D. Rosenblum. A comparative study of coarse- and fine-grained safe regression test selection. *ACM TOSEM*, 10(2):149–183, Apr. 2001.

[4] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[5] H. Cleve and A. Zeller. Finding failure causes through automated testing. In *Proceedings of the Fourth International Workshop on Automated Debugging*, Aug. 2000.

[6] S. Ducasse, M. Lanza, and S. Tichelaar. The moose reengineering environment. *Smalltalk Chronicles*, Aug. 2001.

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

[8] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *20th International Conference on Software Maintenance (ICSM 2004)*, 2004.

[9] R. 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.

[10] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326. ACM Press, Nov. 1997.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, 2001.

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

[13] I. Moore. Jester – a junit test tester. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*. University of Cagliari, 2001.

[14] A. Parrish, J. Jones, and B. Dixon. Extreme unit testing: Ordering test cases to maximize early testing. In M. Marchesi, G. Succi, D. Wells, and L. Williams, editors, *Extreme Programming Perspectives*, pages 123–140. Addison-Wesley, 2002.

[15] L. Renggli. Smallwiki: Collaborative content management. Informatikprojekt, University of Bern, 2003.

[16] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Proceedings ICSE-24*, pages 230–240, May 2002.

[17] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[18] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings ICSM 1999*, pages 179–188, Sept. 1999.

[19] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 230–238, Nov. 1997.

[20] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, SE-28(2):183–200, Feb. 2002.