An Evaluation of Change-Based Coverage Criteria

Marc Fisher II*
Virgina Tech
Blacksburg, VA, USA
fisherii@google.com

Jan Wloka
IBM Rational Research Lab
Zurich, Switzerland
jan_wloka@ch.ibm.com

Frank Tip
IBM Research
Yorktown Heights, NY, USA
ftip@us.ibm.com

Barbara G. Ryder
Virginia Tech
Blacksburg, VA, USA
ryder@cs.vt.edu

Alexander Luchansky
Vanguard Group, Inc.
Malvern, PA, USA
alexander_luchansky@vanguard.com

ABSTRACT

Various coverage criteria are commonly used to assess the quality of test suites, but achieving full coverage according to these criteria is often impossible or impractical. Our research starts from the popular assumption that a disproportionate number of faults is likely to reside in recently changed code. Based on this assumption, we propose several *change-based coverage criteria* that reflect to what extent changes with respect to a previous program version are exercised by a test suite. In a set of experiments on programs from the SIR repository, we found change-based criteria to reveal faults better than traditional criteria, and to enable the construction of much smaller test suites with similar fault detection effectiveness. We also report on a case study that shows that achieving 100% coverage according to a change-based criterion is feasible and that by doing so we were able to find additional faults, including one fault that was not intentionally seeded in the subject program.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and debugging

General Terms

Measurement, Reliability

1. INTRODUCTION

Software evolves throughout its lifetime as developers fix faults, adapt applications to changing requirements and add new functionality. By running a test suite after their edits, developers can expose faults and ensure that there are no unexpected consequences introduced by their changes. A high quality test suite should support this validation of developer changes. If a test suite runs successfully after an edit, this may not suffice, because a successful test does not imply the absence of erroneous program behavior, but rather its inability to expose any faults. Therefore, there is a need to assess the quality of a test suite with regard to exposing possible faults introduced by program changes.

Traditional code coverage criteria measure the percentage of specific code constructs (e.g., statements, methods, or branches [10]) executed by a particular test or test suite. The use of code coverage was an early systematic software testing technique [9]. The basic assumption is that a test suite is effective at revealing faults if it exercises the code where the fault is located. Therefore, increased code coverage is expected to correlate with more revealed faults.

Two important properties are desirable for a practical coverage criterion for a test suite: fault detection ability and usability. A criterion's *fault detection ability* is a measure of the strength of the correlation between the coverage achieved and the number of faults exposed under that criterion. A criterion's *usability* refers to how feasible it is for developers to add tests that cover program constructs that are uncovered according to that criterion. Given a criterion with both properties, raising the level of coverage indicates a higher quality test suite and increases developer confidence in the correctness of the code.

While in practice various criteria are good at fault detection, they often lack the usability property. Traditional criteria consider coverage across the entire program for which high coverage is difficult to achieve. This forces developers to estimate the importance of covering certain constructs rather than others, especially in the context of unfamiliar code. Moreover, since 100% coverage is considered infeasible, organizations often select some arbitrary coverage level (e.g., 70%) as their target [4].

Because developers are concerned with testing the effects of code changes, a coverage criterion should focus on the changes a developer has made, and indicate which tests are needed to exercise those changes, including all their possible program behavior effects. In this work we define a family of *change-based coverage criteria*, that incorporate semantic knowledge about the impact of an edit on a program's behavior.

The research in this paper makes the following contributions:

- Formal definition of several change-based code coverage criteria for object-oriented applications.
- The first comparison of change-based coverage criteria for object-oriented applications with respect to the ability of different test suites to reveal regression faults. Empirical results using three programs from the SIR repository [5] indicate that change-based criteria reveal faults better than traditional criteria, and result in smaller test suites with similar fault detection properties.

*Current Affiliation: Google, Mountain View, CA, USA

PASTE'11, September 5, 2011, Szeged, Hungary.

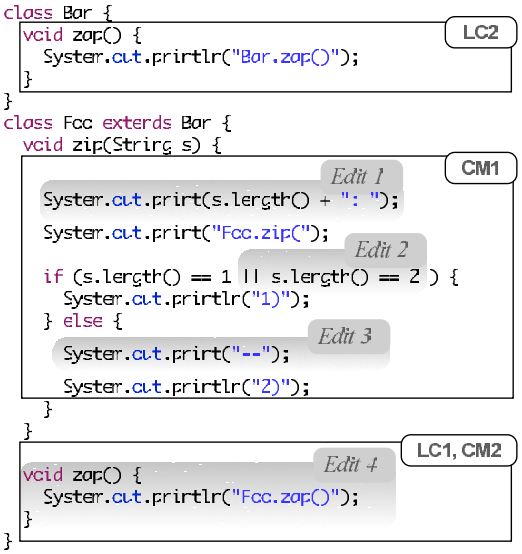

Figure 1: Original and Edited Version of the Example Program with Edits and Atomic Changes (Textual edits are shown with shaded boxes. Atomic changes (CM, LC) are indicated by framed boxes. The original version of the program excludes all shaded code fragments, and the edited version can be constructed by adding these fragments to the program.)

- A case study on one of the benchmarks demonstrates that a change-based criterion can define an achievable testing goal of 100% coverage.

2. COVERAGE CRITERIA

Code coverage is a white-box testing technique to measure how much of a program is exercised by a test suite. Each test in the suite uses assertions to specify expected program behavior. A run of a test suite tries to verify that the program works as expected. Each run computes the degree to which the program is covered by tests as measured by a specific coverage criterion, which is defined in terms of a measurable unit, such as a method, statement, or branch. In this section, we give an overview of all the coverage criteria compared in our empirical study.

2.1 Illustrating Example

Figure 1 shows two versions of a small example program that we will use to illustrate several traditional and change-based coverage criteria. The original version contains the lines of code that lie outside of the shaded areas. The edited version can be obtained by adding all code in the shaded boxes. *Edit 1* adds a new print statement, *Edit 2* adds a test to the `or` condition in the `if` statement, *Edit 3* adds a new print statement to the `else` branch, and *Edit 4* adds a method `Bar.zap()` that overrides `Foo.zap()`. The four atomic changes computed from the textual edits and labeled on the right side of Figure 1 are defined in Section 2.3. For each coverage criterion, we discuss how a test suite can be constructed that attains full coverage of the program, summarizing each test suite simply by the set of method calls that it performs.

2.2 Traditional Coverage Criteria

These criteria are measured with respect to the complete program. Method, statement and branch coverage are commonly used.

Method Coverage.

Method (or function) coverage [25] is commonly used to measure which methods (functions) in the program are invoked during test execution. It indicates whether at least one statement within a method is executed at least once during a test suite run. Method coverage does not differentiate the individual statements actually executed. If we consider the edited version of the example program, 100% method coverage can be achieved by running a test suite containing the following method calls:

```
new Foo().zip(""); new Foo().zap();
new Bar().zap();
```

Statement Coverage.

Another common coverage measure is statement (or line) coverage [10]. Statement coverage reports all statements that are invoked at least once during a test suite run. Statement coverage subsumes method coverage, (i.e., a test suite with 100% statement coverage will have 100% method coverage). A test suite with the following method calls reports 100% statement coverage in the example program

```
new Foo().zip(""); new Foo().zip("a");
new Foo().zap(); new Bar().zap();
```

The extra call to `new Foo().zip("a")` exercises the statement on the `then` branch of the `if` statement, which was not covered by the previous test suite for method coverage.

Branch Coverage.

Branch coverage [10] requires that each outcome (e.g., *true* and *false*) of each branch in a logical expression controlling an `if`, loop, or `switch` statement be covered. For practical reasons (we use instrumentation tools that operate at the bytecode level), we assume that composite logical expressions are decomposed into a sequence of lower-level branches. For example, using short circuit execution, full branch coverage of a logical `or` expression *a* `or` *b* can be obtained with: (i) *a* ==T, (ii) *a*==F, *b*==T and (iii) *a*==F, *b*==F. Note that the last case is required to be covered, but the settings *a*==T, *b*==F and *a*==T, *b*==T are not.

As usually defined, branch coverage subsumes statement coverage [14]. To maintain this constraint, in our implementation we require branch coverage to cover all methods in addition to all decisions within these methods. A set of method calls that achieves 100% branch coverage on the example is:

```
new Foo().zip(""); new Foo().zip("a");
new Foo().zip("ab"); new Bar().zap();
new Foo().zap();
```

2.3 Change-based Coverage Criteria

The basis for the change-based coverage criteria used in our empirical study is a previously developed change impact analysis implemented in the tool CHIANTI [19, 15, 23]. This analysis computes an abstract representation of a textual difference between two program versions and decomposes it into a set of atomic (or smallest) changes to a program. The resulting representation enables a classification of different kinds of changes and their dependences, making textual edits amenable to program analysis. Various change categories are supported, such as change method (CM), add field (AF), and lookup change (LC) (i.e., a change to dynamic dispatch) [15]. However, in this paper we use a simplified version of the change model including only change method (CM) changes and lookup (LC) changes.

Atomic Change Coverage.

The *atomic change coverage* criterion reflects the CM and LC atomic changes that are exercised by a test suite. CM changes correspond to methods whose code has changed. CM changes also summarize other changes; for example, adding a new instance field to a class implicitly changes the constructor method for that class to initialize an additional field. Note that we generate one CM change per method regardless of the number of changed statements within the respective method's body, (e.g., CM1 in the example represents *Edits 1 – 3* as a single CM change).

A lookup change (LC) represents the effect of an edit on dynamic dispatch. It is represented as a pair of the form $<C, f()>$ indicating that the behavior of invoking a method $f()$ on an object of runtime type C has changed. Many kinds of edits may alter the existing dynamic dispatch behavior of a Java program, such as adding the overriding method `zap()` to class `Foo` in our example (LC1, LC2), or changing method visibility from *private* to *public* [15, 19, 23]. A test suite with the following method calls covers all CM and LC changes and achieves 100% atomic change coverage on the example program:

```
new Foo().zip(""); new Foo().zap();
new Bar().zap();
```

Changed Method Coverage.

The *changed method coverage* criterion reports how many of the CM changes are covered by a test suite. It uses the same approach as atomic change coverage but is restricted to CM changes. As such, changed method coverage is subsumed by atomic change coverage. The following method calls within a test suite result in 100% changed method coverage:

```
new Foo().zip(""); new Foo().zap();
```

Changed Statement Coverage.

A test suite's *changed statement coverage* reflects the set of statements that were changed or added since the previous version and exercised by at least one test in the suite. A test suite with the following method calls reports 100% changed statement coverage:

```
new Foo().zip(""); new Foo().zap();
```

A change impact analysis at the level of individual statements (or lines) implemented by JDIFF [1] is used to compute changed statement coverage. Two different versions of a program are compared and both differences and correspondences at statement-level are identified to capture all the changed statements in a Java program.

Changed Branch Coverage.

There are multiple possible ways to define a changed branch coverage criterion. For example, one could focus only on those branches where the condition has changed, or one could also include branches where the body has changed. Since we are not aware of any existing definition or approach for identifying a set of changed branches within an application, we define a simple, conservative notion of changed branch coverage based on the set of CMs identified by CHIANTI. Our *changed branch coverage* criterion requires that all branches within each changed method are covered, even if none of the edits affected the corresponding control statement. In addition, the criterion requires that all CM changes are covered, in order to ensure that changed branch coverage subsumes changed statement coverage. According to this criterion, a test suite with the following method calls achieves 100% changed branch coverage:

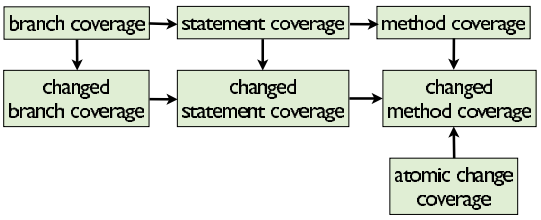

Figure 2: Subsumption relationships between the criteria.

```
new Foo().zip(""); new Foo().zip("a");
new Foo().zip("ab"); new Foo().zap();
```

2.4 Subsumption Relationships

Figure 2 shows the subsumption relationships that hold between the coverage criteria discussed in this section. The presence of an arrow from a criterion *A* to a criterion *B* in the figure means that 100% coverage according to criterion *A* implies 100% coverage according to criterion *B*. Note that atomic change coverage is not subsumed by any other coverage type as it is the only criterion to consider the coverage of (changed) method dispatch behaviors.

3. EXPERIMENTAL EVALUATION

In order to better understand the properties of the different coverage criteria, we conducted a study that compares these criteria by answering the following research questions:

RQ1 How well do the different coverage criteria predict the relative quality of test suites?

RQ2 How do test suites constructed according to the different coverage criteria compare in terms of size and number of regression faults exposed?

3.1 Coverage Criteria

Our empirical study compares the effectiveness of the seven different code coverage criteria defined in Section 2: *method* coverage, *statement* coverage, *branch* coverage, *changed method* coverage, *changed statement* coverage, *changed branch* coverage and *atomic change* coverage. The first three of these are traditional code coverage metrics in common use. To identify the set of changed statements empirically for each program version, we used the JDIFF tool [1]. COBERTURA, version 1.9 [6] was used to run each test suite and to measure method, statement, branch, changed statement, and changed branch coverage. We used JUNITMX [23] to identify atomic changes, run each test suite, and calculate changed method and atomic change coverage as well as the faults exposed by each suite.

3.2 Benchmarks

To evaluate our technique, we used three existing Java applications, *JMeter*, *JTopas*, and *NanoXML*, available from the SIR repository [5]. The SIR repository contains between four and six versions for each of these applications, with an associated test suite for each version, and with associated seeded faults for most versions. Since this study is focused on changes to the software, we used successive version pairs of each application from the SIR repository as shown in column 2 of Table 1, resulting in 12 different version pairs.[1]

[1]Version 4 of *NanoXML* did not include any seeded faults so we were unable to use the v3/v4 version pair of this application

		meths		stmts		brchs		chg meths		chg stmts		chg brchs		atomic chgs	
benchmark	**version**	**all**	**cov**	**all**	**cov**	**all**	**cov**	**all**	**cov**	**all**	**cov**	**all**	**cov**	**all**	**cov**
JMeter	v0/v1	3082	1222	16993	6865	7042	2059	897	337	5761	2451	2701	641	1974	534
	v1/v2	3033	1192	16948	7036	7186	2029	281	119	2408	1276	1156	236	1209	401
	v2/v3	3650	1591	18412	7720	8541	2598	1530	874	8605	4760	4121	1320	5294	2483
	v3/v4	3767	1666	19038	8050	8635	2679	370	145	3163	1462	1062	253	1046	331
	v4/v5	3849	1701	20519	8571	8677	2735	159	74	2069	766	826	204	247	101
JTopas	v0/v1	199	164	796	683	625	434	30	27	185	157	60	46	55	49
	v1/v2	213	178	878	744	726	490	46	41	223	175	154	90	79	70
	v2/v3	490	349	2181	1620	1692	1025	360	245	1404	984	1242	732	797	549
NanoXML	v0/v1	120	98	925	717	552	376	68	52	645	471	416	266	110	85
	v1/v2	177	106	1100	745	625	394	126	69	684	428	411	260	242	97
	v2/v3	214	126	1391	921	810	487	78	54	651	446	334	204	187	115
	v4/v5	229	134	1509	981	876	508	64	50	590	375	420	279	84	59

Table 2: Criteria Details

		tests		faults	
benchmark	**version**	**all**	**used**	**all**	**rev**
JMeter	v0/v1	77	77	20	5
	v1/v2	80	80	21	5
	v2/v3	78	77	20	8
	v3/v4	78	75	13	1
	v4/v5	97	80	13	2
JTopas	v0/v1	126	126	10	7
	v1/v2	128	128	12	6
	v2/v3	209	206	17	8
NanoXML	v0/v1	214	214	7	7
	v1/v2	214	214	7	6
	v2/v3	216	216	11	10
	v4/v5	216	216	9	9

Table 1: Benchmark Details

Column 3 of Table 1 indicates the number of tests available for the most recent version of each version pair. COBERTURA required that each test be run in isolation from the rest of the tests in the test suite; however, some tests for *JMeter* v3, v4, and v5 could only be successfully executed when run as part of the entire test suite. Therefore, we removed these tests from the test suites that we used for this study; column 4 shows the number of tests actually used. The faults columns of Table 1 show the total number of seeded faults for each version pair (all) and the number revealed by the pool of tests we used for each version (rev). Table 2 provides the number of coverable elements (all) and the number of those elements covered by the test suite used (cov) for each of the test coverage criteria we considered.

3.3 Experimental Methodology

For each version pair, V_{old} and V_{new}, we created faulty versions of V_{new} by enabling a single fault within V_{new}. Then, we computed coverage and fault detection matrices by executing each test case t in the test suite on each faulty version of V_{new} as well as on the correct version of V_{new}. For each t, we created coverage and fault detection matrices that indicated the coverable elements covered by t and the faults exposed by t, respectively. We considered an element covered by the execution of t if for at least one version of V_{new} (correct or faulty) that element was covered. A fault f was exposed by test t if t failed on the version of V_{new} with that fault f enabled. The coverage values for a test suite were calculated by unioning the matrix entries for their constituent test cases.

3.4 RQ1

The goal of RQ1 was to determine which of our coverage criteria most accurately measures the relative quality of different test suites with respect to exposing regression faults. To answer this question, we needed to study the effect of varying coverage on fault exposure by measuring the coverage and fault exposure on several test suites for each VUT (i.e., version pair under test). Since we only were supplied with a single test suite for each version, we needed to construct additional test suites. This was accomplished by randomly selecting 100,000 different subsets of the original test suite for each VUT. Note that we only needed to run each test in the original test suite once. After randomly constructing new test suites, we used the previously collected coverage and fault exposure data to calculate the coverage and exposed faults for each test suite.

Table 3 shows the correlation coefficients computed for coverage vs. fault detection for each of the coverage criteria on each of the VUTs. The coefficients were computed using the Kendall τ method [8]. Kendall τ is a standard test that computes the correlation between the ranks of the values for two variables. Kendall τ is 1 if the ranks of the values for the two variables exactly match, -1 if the ranks of the values are in exactly the opposite order, and near 0 if the variables are not related. The correlation coefficients of different coverage criteria can be compared to determine which criterion more accurately predicts the relative strengths of the different test suites at exposing faults. Specifically, if criterion C_1 correlates more strongly with fault detection (i.e., has a higher correlation coefficient) than criterion C_2, then C_1 is the better criterion for predicting which of two test suites is more likely to reveal regression faults. It should be noted, however, that this says nothing about whether a particular test suite with high C_1 coverage is more likely to reveal regression faults than another test suite with equally high C_2 coverage.

For 7 of the 12 version pairs, changed branch coverage showed the strongest correlation with fault exposure. These seven cases included all version pairs of *JMeter* and two of the three *JTopas* pairs. Looking at the cases where changed branch coverage was not the winner, the primary difference seems to be the percentage of all of the branches that were identified as changed. For example, on *JTopas* v2/v3, where changed statement coverage won, 73% of all of the branches were identified as changed. In contrast, on *JTopas* v1/v2, where changed branch coverage won, only 21% of the branches were identified as changed. In general, the five runs where changed branch coverage was not the best had a higher proportion of changed branches (and changed methods and changed statements) than the seven runs where changed branch coverage

benchmark	version	meths	stmts	brchs	chg meths	chg stmts	chg brchs	atomic chgs
JMeter	v0/v1	0.5180	0.5237	0.5241	0.5075	0.5203	**0.5439**	0.5073
	v1/v2	0.4762	0.4685	0.4798	0.5438	0.5284	**0.5922**	0.4801
	v2/v3	0.5350	0.5326	0.5415	0.5392	0.5336	**0.5449**	0.5211
	v3/v4	0.4386	0.4417	0.4451	0.5073	0.4389	**0.5416**	0.4359
	v4/v5	0.5183	0.5200	0.5276	0.5740	0.5669	**0.6767**	0.5884
JTopas	v0/v1	0.6767	0.7081	0.6631	0.7443	0.7343	**0.7826**	0.7376
	v1/v2	0.6110	0.6783	0.6931	0.6637	0.7516	**0.7968**	0.6903
	v2/v3	0.6487	0.6728	0.6634	0.6555	**0.6799**	0.6453	0.6472
NanoXML	v0/v1	0.5450	0.6516	0.5917	0.5078	**0.7422**	0.6534	0.5078
	v1/v2	0.7509	0.6792	0.7040	0.7353	0.6821	0.7095	**0.7529**
	v2/v3	**0.8093**	0.7693	0.7917	0.7313	0.7792	0.7736	0.7976
	v4/v5	0.5505	0.5697	0.5430	0.3286	**0.7254**	0.5137	0.3286

Table 3: Correlation Coefficients (**bold** indicates best value for row, all values were statistically significant ($p < 0.05$))

was the best (although there was some overlap in the ranges). This suggests that the size of the change as a proportion of the entire application is an important component in determining the effectiveness of using a change-based coverage criterion. This is not unexpected as the main purpose of change-based criteria is to focus attention on testing the changed portions of the application, and, in cases where large portions of the code have changed, it is reasonable to expect that such a focus would be less effective. However, it should be noted that in four of the five cases where changed branch coverage was not the best, another change-based criterion performed the best, suggesting that even with large sets of changes a change-based criterion could be useful in guiding testing effort.

Another way to look at these results is by comparing the correlation coefficient for traditional coverage criteria to those for their change-based counterparts. Comparing the results for *method* coverage to those for *changed method* coverage, we see higher correlation coefficients for the latter on 7 of the 12 version pairs. The *changed statement* coverage criterion outperformed the traditional *statement* coverage criterion in 10 of 12 cases, and the *changed branch* criterion did better than the traditional *branch* coverage criterion in 9 of 12 cases.

These results suggest that by focusing attention on changes, we can better indicate the relative quality of test suites in exposing regression faults. This result may not seem surprising as the faults are in the changed portions of the applications; therefore, executing these portions of the programs is likely to execute the faulty statements. However, it is quite possible that the context in which a fault is executed is just as important as executing the fault itself. Therefore, a criterion that emphasizes exercising more of the program may be a better indicator of the regression fault detection effectiveness of a test suite.

Summary. We found that:

- change-based criteria provide better predictions about the relative regression fault detection exposure of test suites than non-change-based criteria;
- of the change-based criteria, changed branch coverage generally provided the best prediction.

3.5 RQ2

Our second research question focused on evaluating the tradeoff between test suite size and fault-detection effectiveness, for test suites created using the different criteria. The ideal evaluation mechanism would have been to generate a variety of adequate test suites for each of the criteria and compare them. Unfortunately, doing so in a fashion that allows fair comparisons of the criteria is expensive and difficult. Therefore, we approached this problem by minimizing the existing test suite for each VUT using each of the coverage criteria.

Since the goal of our minimization was to select subsets of the original test suite without sacrificing coverage, an optimal or near-optimal minimization technique was not required. Instead, we implemented a simple greedy minimization technique that allowed us to readily generate different test suites with no completely redundant tests. Our minimization technique begins with an empty test suite, S. Then, we incrementally add a test to S by randomly selecting a test for the VUT that increases the coverage of S. This step is repeated until the coverage achieved by S equals the coverage of the entire test suite for the VUT. We then iterate through the tests in S, identifying and removing any test whose removal does not decrease the coverage of S. While this approach does not guarantee a minimal test suite, in practice we found that it effectively generated small test suites that varied little in size and included different tests.

For our experiments, we generated 20 unique test suites for each criterion, VUT pair. For each of these test suites, we calculated the number of faults exposed and the test suite size. The averages of these two metrics across the 20 test suites is presented in Table 4.

Although method coverage minimized suites revealed more faults than changed method coverage minimized suites on eight of the version pairs, this difference was small, only exceeding one fault on four of the version pairs. Interestingly, this increase in fault exposure comes at considerable expense. The test suites for method coverage tended to be significantly larger than the test suites for changed method coverage, ranging from 1.04 times as large to 5.67 times as large, and exceeding 2.0 times as large on 7 of the 12 version pairs. The same pattern was found when comparing statement coverage minimized suites with changed statement coverage minimized suites and branch coverage minimized suites with changed branch minimized suites, although the differences in number of exposed faults and in test suite size tended to be smaller. These findings show that a significant benefit in efficiency can be gained by focusing on testing changes while missing few regression faults.

Comparing the different change-based criteria to each other, we see that changed statement coverage minimized suites reveal more faults and contain more tests on average than the other change-based criteria minimized suites and changed method coverage minimized suites generally reveal fewer faults and contain fewer tests than the other minimized suites. There is not a consistent difference in ranking between changed branch minimized suites and atomic change minimized suites. Thus, for change-based coverage crite-

		meths		stmts		brchs		chg meths		chg stmts		chg brchs		atomic chgs	
object	**version**	**size**	**rev**	**size**	**rev**	**size**	**rev**	**size**	**rev**	**size**	**rev**	**size**	**rev**	**size**	**rev**
JMeter	v0/v1	21	3.35	36	3	36	3	15	3.1	27	3	22	3.33	17	3.2
	v1/v2	23	5	38	5	36	5	11	5	23	5	13	5	16	5
	v2/v3	28	8	37.8	8	34	8	27	8	33.8	8	28	8	30	8
	v3/v4	26	1	36	1	32	1	9	1	22	1	10	1	9	1
	v4/v5	26	2	36	2	33	2	11	1.05	20	2	14.7	1.65	12	2
JTopas	v0/v1	15	7	21	7	21	7	6	5.2	14.1	7	6	7	6	5.3
	v1/v2	15	6	22	6	22	6	8	6	12	6	6	5	10	6
	v2/v3	30	4.5	45	8	46.65	8	20	4.1	35	7.5	19.6	7	20	4.15
NanoXML	v0/v1	15	5.7	25.5	7	21	6.45	5	3.55	11.95	6.25	10	5.55	5	3.8
	v1/v2	16	4.7	25.5	5.25	21	4.3	10.2	4.65	14.55	5.25	11	4.25	10	4.7
	v2/v3	17	6.5	29	10	24	9	10	3.6	22	10	15	9	13	6.5
	v4/v5	17	7.4	29	9	23	7.7	3	4.25	10.55	6.35	7	7	3	3.5

Table 4: Average Size of and Faults Revealed by Minimized Test Suites

ria, there is the usual trade-off between fault-exposure potential and test suite size.

Summary. We found that:

- test suites constructed according to change-based criteria found approximately the same number of faults on average, but were significantly smaller than test suites constructed according to corresponding non-change-based criteria;
- there is a trade-off between fault-exposure potential and test suite size for test suites constructed according to different change-based criteria.

4. CASE STUDY: JTOPAS

In this section we report on a case study investigating the feasibility of achieving 100% change coverage using the atomic change criterion. We chose this criterion for three reasons:

- It provides information (computed using program analysis) about semantic changes in object-oriented programs that can guide the developer in writing new tests (e.g., the addition of a method resulting in a new target of a dynamic dispatch);
- Since this criterion tracks method-level changes, there are fewer changes to cover than, for example, with statement coverage, making 100% coverage a more feasible goal; and
- We have a long-standing interest in analyzing the impact of changes in a object-oriented setting [15, 19, 23].

As the subject program for this case study, we chose *JTopas*, a medium-sized Java application for which several versions with associated test suites are available from the SIR repository [5]. Moreover, the SIR repository provides a set of seeded faults to simulate regression failures. Our technical report [7] presents a similar case study on *NanoXML*, another program from SIR.

To demonstrate the effectiveness of the atomic change criterion, we ran the test suites for each changed version of *JTopas* and then tried to write additional tests to cover all changes reported as uncovered. Although our results are anecdotal, we believe they demonstrate that the atomic change criterion aided the construction of new tests and thereby, the achievement of 100% change coverage.

Quantitative Results. Table 5 summarizes our quantitative results. For each version pair the table shows: (i) the number of uncovered changes, (ii) the number and LOC of additional tests written, and (iii) the number of additional seeded and real (i.e., non-seeded) faults exposed by the additional tests.

	uncovered	added tests		#faults exposed	
version	**changes**	**#**	**LOC**	**seeded**	**non-seeded**
v0/v1	5	3	91	1	0
v1/v2	7	2	74	0	0
v2/v3	106	31	104	2	1

Table 5: Quantitative Results of *JTopas* Case Study

For example, for the v2/v3 version pair of *JTopas*, 31 additional tests were written, comprising a total of 104 lines of source code, in order to cover CM and LC changes reported as uncovered by our tool. *These additional tests revealed 2 additional seeded faults, as well as a real fault that was accidentally introduced by the developers as they made their changes* (a more detailed discussion follows below).

For *JTopas*, newly written tests exercising the uncovered CM changes usually also covered any reported LC changes. Hence, the use of the changed method coverage criterion would have produced similar results in terms of the required effort and the number of exposed faults. To a large extent this is because *JTopas* is not written in a very object-oriented style. Therefore, we suspect that different subject programs with more complex class hierarchies might have yielded different results.

Test Creation Process. Initially, we were unfamiliar with the code of *JTopas*. After some initial experience with trying to cover methods that were previously not exercised by the test suite, we quickly realized that developing new test cases "from scratch" can be quite challenging. As a result, we converged on a process for deriving new tests from existing ones. Specifically, we search for a covered method m containing a call to a changed method m' that we need to cover (if we are unable to do this, we try to find a covered method m that indirectly calls m'). Presumably, the call to m' is not executed because a control condition is not satisfied (e.g., the call to m' is on an unexecuted else-branch of an if-statement). In such cases, the challenge is to understand how—by changing the program input—we can persuade the program to select the branch containing the call to m'. In the absence of automated tool support for this task, we had to inspect the program state in the debugger and understand how the conditions that guard the call to m' depend on program inputs or constant values. Following this approach, we were able to cover all but a few of the atomic changes with relatively little effort.

The use of a change-based criterion helped us to focus on new tests covering the edited portions of the program. In contrast, to achieve 100% coverage with a non-change-based criterion, we would

have introduced unnecessary tests for portions of the program unaffected by the edit. Additionally, by reporting changes at the method-level, the atomic change criterion suggested a natural structure for the additional tests that statement-based criteria did not.

We estimate that the amount of effort involved in writing the additional tests was in the order of 2-8 hours per version pair; however, we conjecture that the original developers could have performed this task in a fraction of the time, given their familiarity with the code.

Experience with JTopas. For *JTopas*, most of the changes were of a trivial nature and could be covered with relatively little effort. For example, in the *JTopas* v1/v2 version pair, an uncovered getter method `PluginTokenizer.getKeywordHandler()` was tested by adding a simple assertion to a test that already invoked the corresponding `setKeywordHandler()` setter method.

A more interesting case in *JTopas* v1/v2 was a new uncovered method `AbstractTokenizer. getTextUnchecked(int,int)`, which is similar to an existing covered method `AbstractTokenizer.getText (int,int)`. The new `getTextUnchecked()` method does not check to make sure that its parameters are within range, and if they are not, the JavaDocs report that *"a java.util.IndexOutOfBoundsException may occur, or uninitialized data may be retrieved"*. Covering this method is a bit involved, because it is not intended to be invoked directly from a test. However, the `getText()` method is invoked by a method `current()`, a method in the same class, which is invoked by several tests, including `TestInputStreamTokenizer.testLineCounting()`. We covered `getTextUnchecked()` by adding: (i) a `currentUnchecked()` method, which is similar to `current()` but calls `getTextUnchecked()` instead of `getText()`, and (ii) a test `testLineCounting2()` that is identical to `testLineCounting()` except that it calls `currentUnchecked()` instead of `current()`. Interestingly, this added test exposed one of the seeded faults for *JTopas* that was not exposed by the original test suite, thus illustrating how the use of a change-based coverage criterion can help improve software quality.

Another interesting case occurred in the *JTopas* v2/v3 version pair. While adding new tests that exercise the uncovered changes, we discovered a bug in the newly added `Token.getEndPosition()` method (i.e., a non-seeded fault). This method is currently implemented as:

```
public int getEndPosition() {
  return getLength() - getStartPosition();
}
```

but it should be:

```
public int getEndPosition() {
  return getLength() + getStartPosition();
}
```

This newly added method was not covered by any of the existing test cases, and the use of atomic change coverage as a test adequacy criterion led us directly to this non-seeded fault.

Discussion. We found the use of atomic change coverage as a test adequacy criterion feasible in terms of the amount of effort involved, and useful for pointing programmers at untested changes that might otherwise be overlooked. The case study revealed 2 previously unexposed seeded faults and 1 real fault, thus providing some evidence that using atomic change coverage as a test adequacy criterion may help prevent errors and improve code quality. We present similar findings for the *NanoXML* case study in our technical report [7].

5. THREATS AND LIMITATIONS

There are a wide range of threats and limitations relevant to our research. The primary threats to validity are external, affecting the generalizability of our results. These include the choice of benchmarks and the versions, test suites and seeded faults for these benchmarks. Any of these choices may not be representative of what is found within the wider software development community and, therefore, our results may not generalize. To mitigate these threats, we used benchmarks from the SIR repository that have been used in the evaluation of a wide variety of testing methodologies. The particular selected benchmarks were chosen based on the ability of the tools we were using to handle them and to cover a range of different applications.

Our choice of benchmarks and study design also limits the applicability and interpretation of our results. In particular, we used sampled subsets of the tests provided for these benchmarks as proxies for different possible test suites. In practice, it is possible that test designers creating tests for a particular test coverage criteria would create different types of test suites than those studied. Additionally, our goal was to study the usefulness of different coverage criteria in finding regression faults in applications. In practice there are also likely to be faults left over from earlier versions of the program under test, and the use of a change-based criterion actually may make it more difficult to find these faults.

An additional threat to validity arises from the differences between the change sets identified by JDIFF and CHIANTI. JDIFF computes changes by analyzing bytecode, while CHIANTI computes changes by analyzing the source code. This leads to differences in the set of methods with changes identified by JDIFF and CHIANTI, breaking the subsumption relationships discussed in Section 2.4. To determine how much impact these differences had on the actual results, we also implemented a changed method coverage based on JDIFF. The differences between the Kendall τ coefficients for the two different changed method implementations were generally small, exceeding 0.1 in only one case (*NanoXML* v4/v5) and 0.01 in 5 other cases, suggesting that the different change analysis techniques had little effect on the results.

6. RELATED WORK

Several different coverage criteria for measuring the adequacy of a test suite have been developed (e.g. [12, 14]), including statement coverage, branch coverage, and various notions of dataflow coverage. In this work, we derive three new criteria, changed statement coverage, changed branch coverage, and changed method coverage, from these traditional coverage criteria, and compare these new criteria to their non-change-based counterpoints.

A number of techniques have been proposed for evaluating different coverage criteria. For example, Weyuker developed a set of 11 properties that a coverage criteria should possess [22] that were later applied to evaluating object-oriented coverage criteria by Perry and Kaiser [13]. Both Wong et al. [24] and Namin and Andrews [11] present studies comparing the effect of test suite size and coverage on fault detection that are similar in setup to ours. Wong et al. find that block coverage correlates better with fault detection than test suite size. Namin and Andrews found that fault detection effectiveness correlated well with block coverage, decision coverage, and two different dataflow coverage criteria; however a model that also included test suite size was even more effective. Unlike our work, these studies are not looking specifically at the practice of regression testing where it is expected that the faults are more likely to be in the changed portions of the code.

Regression testing focuses on ensuring that modifications do not impact pre-existing functionality. Most of the regression testing

literature falls into one of two categories: regression test selection (surveyed in [17]) and regression test prioritization (e.g. [18, 21]). Regression test selection attempts to select a subset of the entire test suite of a program that will identify any faults introduced by the modification into the pre-existing functionalities of the system. Regression test prioritization uses heuristics to reorder the tests in a test suite to increase the likelihood that any newly introduced faults will be revealed earlier in the testing process. These techniques focus on ways of more efficiently using an existing test suite to test a modified version of a system while change-based coverage criteria attempt to evaluate the quality of a test suite relative to a modification and to indicate portions of the modified application that may need additional testing.

Bates and Horwitz [3] and Harrold and Rothermel [16] present techniques for identifying the set of changed dataflow and control-flow testing requirements for a modified program. Their techniques support several criteria including statement coverage, branch coverage, and def-use testing. These techniques use program dependence graph or system dependence graph representations for the original and modified programs to identify the changed testing requirements. While not presented as such, these techniques could form the basis for additional change-based coverage criteria.

The goal of the MATRIX [2] and MATRIXRELOADED [20] projects has been to identify testing requirements for changed software based on data- and control-flow chains and on the symbolic state of the program at the beginning and end of these chains. Then, test inputs are generated to cover these requirements. These coverage criteria have been shown to be more effective and sometimes more cost-effective at identifying changed behavior in the modified programs than changed statement coverage or weaker changed dataflow coverages. However, this work has not directly evaluated the ability of their coverage criteria to expose faults or predict the fault exposure capabilities of different test suites.

7. CONCLUSIONS

Code coverage criteria are commonly used to assess the quality of test suites. The basic idea is that a test suite is likely to be effective at revealing faults if it exercises the code where the fault is located. Therefore, increased code coverage is expected to correlate with more revealed faults. However, achieving full coverage according to traditional coverage criteria is often impossible or impractical when applications contain unreachable code, which may arise due to programmer carelessness, or when the implementation of a feature is incomplete.

In this paper, we start from the popular assumption that a disproportionate number of faults is likely to reside in recently changed code. Based on this assumption, we propose several *change-based coverage criteria* that reflect to what extent changes with respect to a previous program version are exercised by a test suite. In a set of experiments on programs from the SIR repository, we found change-based criteria to reveal faults better than traditional criteria, and to enable the construction of much smaller test suites with similar fault detection effectiveness than those constructed according to traditional coverage criteria. We also reported on a case study that shows that achieving 100% coverage according to a change-based criterion is feasible and that by achieving 100% we were able to find additional faults, including one fault that was not intentionally seeded in the subject program.

8. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDIFF: A differencing technique and tool for object–oriented programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[2] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. MATRIX: Maintenance-oriented testing requirements identifier and examiner. In *Proc. Testing: Academic and Industrial Conference - Practice And Research Techniques*, pages 137–146, 2006.

[3] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. POPL'93*, pages 384–396, 1993.

[4] S. Berner, R. Weber, and R. K. Keller. Enhancing software testing by judicious use of code coverage information. In *Proc. ICSE'07*, pages 612–620, 2007.

[5] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[6] M. Doliner. COBERTURA. `http://cobertura.sourceforge.net/`.

[7] M. Fisher II, J. Wloka, F. Tip, B. Ryder, and A. Luchansky. An evaluation of change-based coverage criteria. Technical Report TR-11-03, Virginia Tech, Mar. 2011.

[8] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[9] J. C. Miller and C. J. Maloney. Systematic mistake analysis of digital computer programs. *CACM*, 6(2):58–63, 1963.

[10] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, Inc, 1979.

[11] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proc. ISSTA'09*, pages 57–68, 2009.

[12] S. C. Ntafos. On required element testing. *IEEE TSE*, 10(6):795–803, 1984.

[13] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *JOOP*, 2(5):13–19, 1990.

[14] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE TSE*, 11:367–375, 1985.

[15] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. CHIANTI: A tool for practical change impact analysis of Java programs. In *Proc. OOPSLA'04*, pages 432–448, 2004.

[16] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proc. ISSTA'94*, pages 169–184, 1994.

[17] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE TSE*, 22:529–551, 1996.

[18] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization. *IEEE TSE*, 27(10):929–948, 2001.

[19] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proc. PASTE'01*, pages 46–53, 2001.

[20] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. ASE'08*, pages 218–227, Sept. 2008.

[21] A. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. Roos. Time-aware test suite prioritization. In *Proc. ISSTA'06*, pages 1–12, 2006.

[22] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *CACM*, 31(6):668–675, 1988.

[23] J. Wloka, B. G. Ryder, and F. Tip. JUNITMX - a change-aware unit testing tool. In *Proc. ICSE'09*, pages 567–570, 2009.

[24] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Proc. ISSRE'94*, pages 230–238, 1994.

[25] Q. Yang, J. J. Li, and D. M. Weiss. A survey of coverage based testing tools. In *Proc. of the International Workshop on Automation of Software Test*, pages 99–103, 2006.