Finding Failure-Inducing Changes in Java Programs using Change Classification

Maximilian Stoerzer
Lehrstuhl Softwaresysteme
University of Passau
Innstraße 32, 94032 Passau, Germany
stoerzer@fmi.uni-passau.de

Barbara G. Ryder and Xiaoxia Ren
Dept. of Computer Science
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854, USA
{ryder,xren}@cs.rutgers.edu

Frank Tip
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
USA
ftip@us.ibm.com

ABSTRACT

Testing and code editing are interleaved activities during program development. When tests fail unexpectedly, the changes that caused the failure(s) are not always easy to find. We explore how change classification can focus programmer attention on failure-inducing changes by automatically labeling changes *Red*, *Yellow*, or *Green*, indicating the likelihood that they have contributed to a test failure. We implemented our change classification tool *JUnit/CIA* as an extension to the *JUnit* component within Eclipse, and evaluated its effectiveness in two case studies. Our results indicate that change classification is an effective technique for finding failure-inducing changes.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids, Testing tools; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Version control

General Terms

Algorithms, Experimentation

Keywords

change impact analysis, debugging, testing, fault localization, version control

1. INTRODUCTION

In modern software development, coding and testing are performed in interleaved fashion to assure code quality. Current development strategies rely heavily on the availability of a test suite to allow a programmer to quickly assess the impact of edits on program functionality. Difficulties occur when testing reveals unexpected behaviors, such as assertion failures or exceptions. Although the programmer knows thereby that she has introduced a bug, she still does not know which part of the edit is responsible for the failure. If the edits are trivially small, it may be easy to find the buggy code. However, as a code base and its test suite grow in size, running the tests after each minor change may become infeasible[1], and the number of changes that occur between successive executions of the test suite is likely to increase. Then, when test failures occur, it may be difficult to isolate the failure-inducing change(s), and tedious manual debugging may be needed.

This paper presents an approach for identifying failure-inducing changes in a system with an associated regression test suite. In contrast to Extreme Programming (XP) [1] where the number of changes between test runs tends to be small, we assume that the size of an edit can become sufficiently large to make the identification of failure-inducing changes a difficult task, and to make automated assistance with this task desirable. Our change classification technique relies on the change impact analysis of [24] to find the tests potentially affected by an edit (i.e., a set of changes), and to associate with each such test, a set of affecting changes. It then classifies these affecting changes as *Red*, *Yellow*, or *Green*, depending on whether they affect (i) tests whose outcome has *improved*, (ii) tests whose outcome has *degraded*, (iii) tests whose outcome has remained *unchanged*, or some combination of (i), (ii), and (iii).

To explore the usefulness of change classification we designed a number of classifiers that assign the colors *Red*, *Yellow*, and *Green* to changes in slightly different ways. Our goal has been to develop classifiers for which *Red* changes are highly likely to be failure-inducing, *Green* changes are highly unlikely to be failure-inducing, and *Yellow* changes fall somewhere in between. With these classifiers we set out to answer the following research questions:

1. Does it work? Can we distinguish failure-inducing changes from other changes through change classification?

2. Which classifier is best? Is there a single change classifier that is always superior to all others, or do different classifiers work better for different applications?

3. If there is no "best" change classifier, is there a set of characteristics of an application that can be used to predict the classifier that will be most effective for it?

To answer these questions, we implemented five change classifiers in a tool called *JUnit/CIA*, an extension of the Eclipse component that integrates the popular *JUnit* testing framework with the

[1] For example, the Eclipse compiler had a test suite of 8803 tests (4830 parser tests + 3973 regression tests) on January 1, 2005. Executing this test suite takes more than 45 minutes on an AMD Athlon64 3200Mhz PC with 2GB RAM.

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.
Copyright 2006 ACM 1-59593-468-5/06/0011 ...$5.00.

Eclipse IDE (see `www.junit.org` and `www.eclipse.org`). The name of the tool reflects the fact that the functionality of *JUnit* is extended with features for Change Impact Analysis. *JUnit/CIA* relies on *Chianti* [24] for: (i) dividing a program edit into its constituent *atomic changes*, (ii) identifying tests *affected* by the edit by correlating (dynamic) call graphs for the tests with the atomic changes, and (iii) determining *affecting changes* for each of these tests. *JUnit/CIA* then classifies changes according to one of the five classifiers and visualizes them using a small extension of *JUnit*'s user-interface. We envision *JUnit/CIA* to be used when running conventional JUnit tests reveals an unexpected test failure. *JUnit/CIA* can be used to compare a current faulty program version to an earlier, successfully tested program version derived either from Eclipse's local history or extracted from a version control repository. *JUnit/CIA* then classifies changes to help the programmer identify the failure-inducing ones. The programmer fixes the problem, and a new successfully tested program version is created.

We conducted two case studies with *JUnit/CIA* to find failure-inducing changes in student programs and in *Daikon* [8], respectively. In each study, we first determined the actual failure-inducing changes by manual examination of the code and then measured the effectiveness of each of the classifiers in identifying those changes. Here, effectiveness is measured by determining how much additional focus on failure-inducing changes is provided by the change coloring, compared to the set of (uncolored) affecting changes reported by *Chianti*. Ideally, we would like to see the failure-inducing changes for a test colored *Red*, and all other affecting changes *Green* or *Yellow*.

In the student programs study, one classifier was superior by correctly focusing programmer attention on the failure-inducing changes in 47.5% of the 444 worsening tests with more than 2 affecting changes while providing misleading information in only a single case. In the *Daikon* study, we studied a pair of versions separated by a total of 6093 atomic changes in which two tests that passed in the original version failed in the edited version. Here, one of the tests was affected by 35 atomic changes, and the other by 34 atomic changes. In this study, a *different* classifier was very effective by focusing the programmer's attention on 4 of the 35 changes for the first test, and on 3 of the 34 changes for the second one. For both of these Daikon tests, the failure-inducing changes were among the few changes that were colored *Red*.

While it seems contradictory that the case studies suggest that different classifiers should be preferred, this is not unexpected in an empirical study. Nevertheless, it is interesting to observe the different characteristics of the code analyzed in the studies to help explain the different outcomes. The student programs study is concerned with the initial development of an application, and is characterized by small differences between versions, and a mixture of improving and worsening tests. In the Daikon study, on the other hand, the application under consideration is more mature, the sets of changes between successive versions is much larger, only a few worsening tests occur, and no improving tests. Therefore, although we make recommendations for when each of the two preferred change classifiers should be used, it is clear that further investigation is needed, and we consider such investigations to be a fruitful topic for future work.

The main contributions of this paper are:

1. We designed a method to identify failure-inducing changes in which changes are classified as *Red*, *Yellow*, or *Green* according to one of several change classifiers.

2. We implemented this method in a practical tool, *JUnit/CIA*, based on *JUnit*, Eclipse, and Chianti [24].

3. We conducted two case studies in which we measured the relative effectiveness of change classification on applications for which we manually identified failure-inducing changes. These case studies indicated that change classification can focus programmer attention effectively on likely sources of failure; however, they were inconclusive with respect to selecting a "best" classifier.

In the remainder of this paper, Section 2 explains our change impact analysis [24] and change classifications intuitively through an example. Section 3 defines the five classifiers used in the case studies presented and interpreted in Section 4. Related work is discussed in Section 5 and conclusions are given in Section 6.

2. EXAMPLE OF OUR APPROACH

Figure 1(a) shows two versions of a small example program. Here, the original version of the program consists of all program fragments *except* for those shown in boxes; the edited version is obtained by adding all the boxed code fragments. Associated with the program are five *JUnit* tests, `testPassPass()`, `testPassFail()`, `testFailPass()`, `testFailFail()` and `testCrashFail()` as shown in Figure 1(b).

We assume that the tests of Figure 1(b) will be used with both the original and edited versions of the program. The name of each test indicates its outcome in each version of the program; for example, `testPassFail()` passes in the original program, but produces an assertion failure in the edited version. By examining the edited program and tests, we can observe that the addition of method `C.zap()` causes the failure of `testPassFail()` and that this is the only test failure due to the edit. Note, the reason for the failure of `testFailFail()` is the same in both versions of the program, namely that `B.bar()` does not have the expected side effect.

Atomic Changes. Our change impact analysis (presented in full detail in [24]) relies on the computation of a set of atomic changes, denoted by $\mathcal{A}$, that captures all source code modifications at a semantic level amenable to analysis. We use a fairly coarse-grained model of atomic changes, with change categories such as added classes (**AC**), deleted classes (**DC**), added methods (**AM**), deleted methods (**DM**), changed method bodies (**CM**), added fields (**AF**), deleted fields (**DF**), and lookup changes (**LC**) (i.e., changes to dynamic dispatch). Regarding changes to method bodies (**CM** changes), note that we generate *one* **CM** change regardless of the number of statements within the respective method's body that have been changed, as we employ a method-level analysis.

Additionally, we compute syntactic dependences between atomic changes. Intuitively, an atomic change A_1 is dependent on another atomic change A_2, if applying A_1 to the original version of the program without also applying A_2 results in a syntactically invalid program (i.e., A_2 is a *prerequisite* for A_1, $A_2 \preceq A_1$). These dependences can be used to construct syntactically valid intermediate versions of the program that contain some, but not all of the atomic changes, as described in detail in [3, 23].

It is important to understand that the *syntactic* dependences do not capture all *semantic* dependences between changes (e.g., consider changes related to a variable definition and a variable use in two different methods). This means that if two atomic changes, A_1 and A_2, affect a given test T, then the absence of a *syntactic* dependence between A_1 and A_2 does not imply the absence of a *semantic* dependence; that is, program behaviors resulting from applying A_1 alone, A_2 alone, or A_1 and A_2 together, *may all be different*.

Figure 2 shows the atomic changes that define the two versions of the example program, numbered 1 through 12 for convenience.

```
public class A {
  public A(int i){ x = i; }
  public void foo(){ x = x + 0; 1 }
  public void bar(){ y = x; 3 }
  public void zap(){ }
  public void zip(){ y = x; 5 }
  public int x;
  public static int y; 4
  public static int getY(){ return y; } 6,7
}
public class B extends A {
  public B(int j){ super(j); }
  public void foo(){ }
  public void bar(){ x++; 2 }
}
public class C extends A {
  public C(int k){ super(k); }
  public void zap(){ x = 5; } 8,9,10,11
}
class D extends A {
  public D(int l) { super(l); }
  public void foo(){ x--; 12 }
}
```

(a)

```
public class Tests extends TestCase {
  public void testPassPass(){
    A a = new A(5);
    a.foo(); a.bar();
    Assert.assertTrue(a.x == 5);
  }
  public void testPassFail(){
    A a = new C(7);
    a.foo(); a.zap(); a.zip();
    Assert.assertTrue(a.x == 7);
  }
  public void testFailPass(){
    A a = new B(8);
    a.foo(); a.bar(); a.zip();
    Assert.assertTrue(a.x == 9);
  }
  public void testFailFail(){
    A a = new B(6);
    a.foo(); a.bar();
    Assert.assertTrue(a.x == 11);
  }
  public void testCrashFail(){
    A a = new D(5); a.foo();
    int i = a.x / (a.x - 5);
    Assert.assertTrue(a.x == 5);
  }
}
```

(b)

Figure 1: (a) Original and edited version of example program. The original program consists of all program fragments *except* those shown in boxes. The edited program is obtained by adding all boxed code fragments. Each box is labeled with the numbers of the corresponding atomic changes. (b) Tests associated with (both versions of) the example program.

Each atomic change is shown as a box, where the top half of the box shows the category of the atomic change (e.g., **CM** for changed method), and the bottom half shows the method or field involved (for **LC** changes, the declaring class and method are shown). An arrow from an atomic change A_1 to an atomic change A_2 indicates that A_1 is dependent on A_2. Consider, for example, the addition of the assignment `y = x` in method `A.zip()`. This source code change corresponds to atomic change 5 in Figure 2. Adding this assignment will lead to a syntactically invalid program unless field `A.y` is also added. Therefore, atomic change 5 is dependent on atomic change 4, an **AF** change for field `A.y`.

In some cases, a single source code change is decomposed into several atomic changes. For example, the addition of `A.getY()` produces atomic changes 6 and 7, where the former models the addition of an empty method `A.getY()`, and the latter the addition of its method body. Observe that atomic change 7 is dependent on atomic change 6, reflecting the fact that a method must exist before its body can be added. Change 7 is also dependent on change 4 (an **AF** change for field `A.y`), because adding the body of `A.getY()` will result in a syntactically invalid program unless field `A.y` is added as well.

The **LC** atomic change category models changes to the dynamic dispatch behavior of instance methods. In particular, an **LC** change $(Y, X.m())$ models the fact that a call to method $X.m()$ on an object of run-time type Y results in the selection of a different method. Consider, for example, the addition of method `C.zap()` to the program of Figure 1. As a result of this change, a call to `A.zap()` on an object of type `C` will dispatch to `C.zap()` in the edited program, whereas it dispatches to `A.zap()` in the original program. This change in dispatch behavior is captured by atomic change 10. Note, **LC** changes also may be generated as a result of a source code change affecting the class hierarchy, such as changing a method from *abstract* to *non-abstract* or from *private* to *public* [24].

Determining Affected Tests. In order to identify those tests that are affected by a set of atomic changes, a call graph is constructed for each test in the *original* program.[2] Our analysis can work with call graphs that have been constructed either using static analysis, or by observing the actual execution of the tests (we use dynamic call graphs in this paper; for details see [24]).

Figure 3 shows the call graphs for the tests of Figure 1(b) in the original program. Edges corresponding to dynamic dispatch are labeled with a pair $< RT, M >$, where RT is the run-time type of the receiver object, and M is the method referenced at the call site. A test is determined to be *affected* if its call graph (in the original program) contains either (i) a node that corresponds to a **CM** (changed method) or **DM** (deleted method) change, or (ii) an edge that corresponds to a **LC** (lookup) change. In Figure 3 clearly all five tests are affected, because they each execute at least one method corresponding to a **CM** change. For example, the call graphs for `testPassPass()` and `testPassFail()` contain nodes corresponding to the changed method `A.foo()` (change 1).

Determining Affecting Changes. In order to compute the set of changes affecting a given test, we construct a call graph for that test in the *edited* program. These call graphs are shown in Figure 4. The set of atomic changes that *affect* a given test includes: (i) all atomic changes for added (**AM**) and changed (**CM**) methods that correspond to a node in the call graph (in the edited program), (ii) lookup changes (**LC**) that correspond to an edge in the call graph, and (iii) their transitively prerequisite atomic changes.

For example, the call graph for `testPassFail()` in Figure 4 contains nodes corresponding to methods `A.foo()`, `C.zap()`, and `A.zip()`. These nodes correspond to atomic changes 1, 9, and 5 in Figure 2, respectively. The call graph for `testPassFail()` also contains an edge labeled `<C,A.zap()>`, corresponding to atomic change 10. From the dependences in Figure 2, it can be seen that change 9 re-

[2] Call graphs contain one node for each method, and edges between nodes to reflect calling relationships between methods.

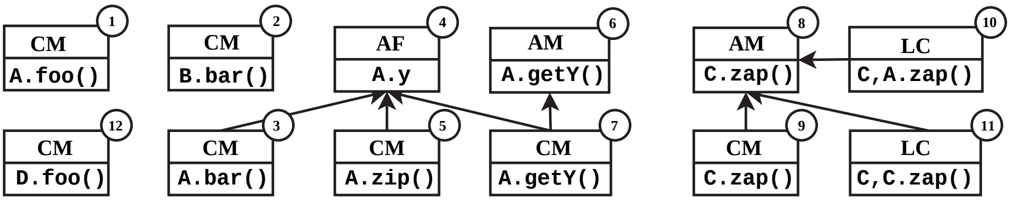


Figure 2: Atomic changes inferred from the two versions of the program.

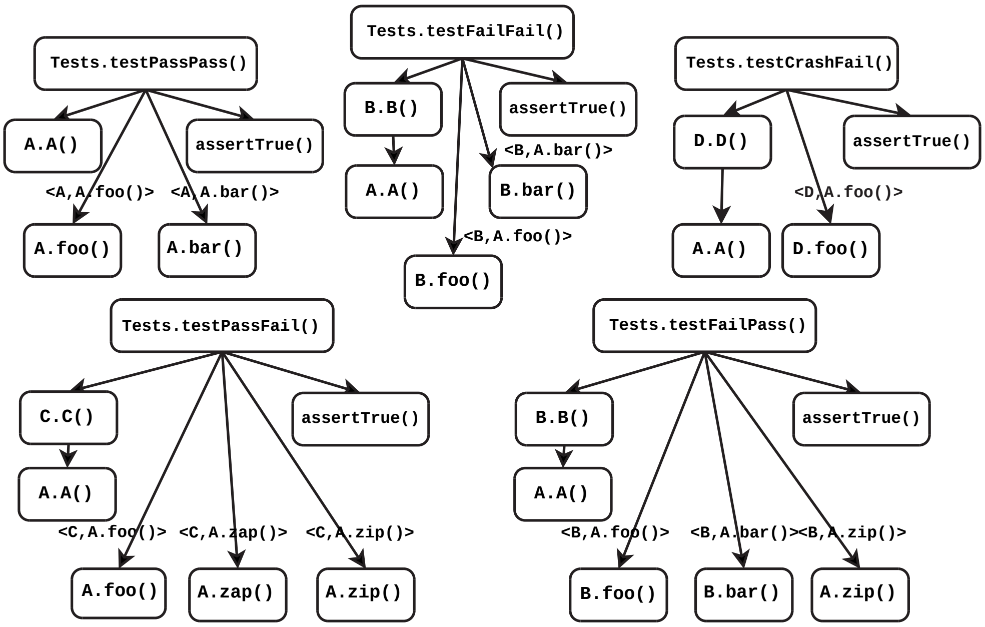


Figure 3: Call graphs for the original version of the program.

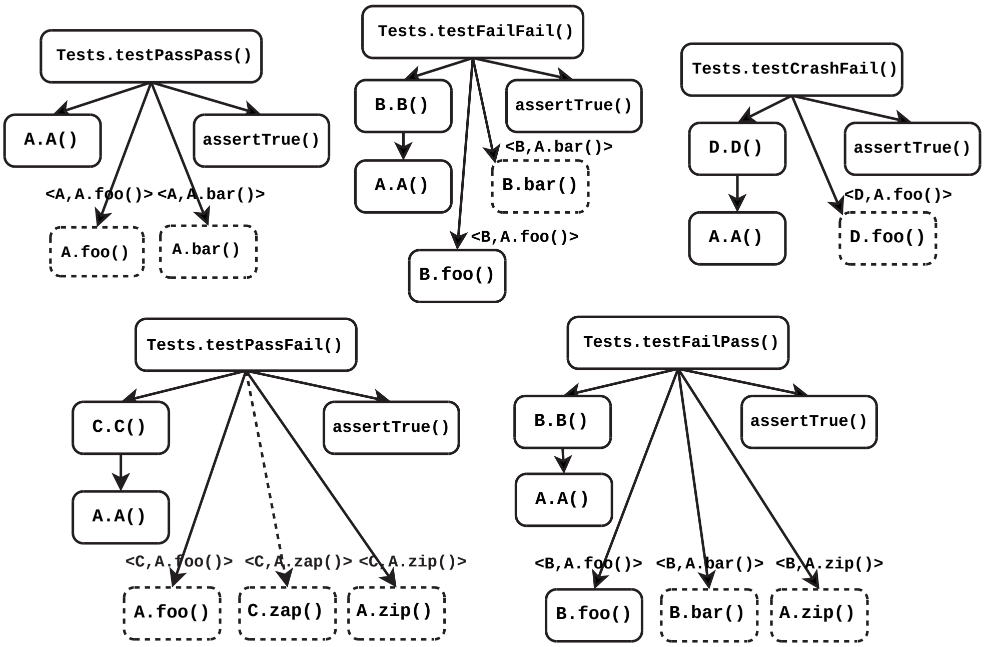


Figure 4: Call graphs for the edited version of the program. Dashed boxes indicate changed/added methods, and dashed arrows indicate changed calling relationships between methods (lookup changes).

quires change 8, and change 5 requires change 4. Therefore, the affecting changes for `testPassFail()` are 1, 4, 5, 8, 9, and 10. Similarly, we determine that 1, 3, 4 are the affecting changes for `testPassPass()`, that 2, 4, 5 are the affecting changes for `testFailPass()`, that only change 2 affects `testFailFail()` and that only change 12 affects `testCrashFail()`.

Change Classification. Thus far, we have seen that there are 12 atomic changes, and that the behavior of each of the five tests is affected by one or more of these changes. The goal of change classification is to answer the following question: *Which of those 12 changes are the likely reason(s) for the test failure(s)?* We provide an answer to this question by classifying the changes according to the tests that they affect. Intuitively, our goal is the following:

- A change that affects only *improving tests*, (i.e., tests such as `testFailPass()` that fail in the original program, but that succeed in the edited version) is classified as *Green*. For example, change 12 (**CM** for `D.foo()`) only affects `testCrashFail()` and thus should be colored *Green*. We consider CRASH to be a worse result than FAIL, because in conducting the experiments described in Section 4, we observed several bugs that resulted in changing the result of a test from FAIL to CRASH.

- A change that affects only *worsening tests*, (i.e., tests such as `testPassFail()` that succeed in the original program, but that fail in the edited version) is classified as *Red*. For example, changes 8, 9, 10 (**AM** and **CM** for `C.zap()` and **LC** for `<C,A.zap()>`) only affect `testPassFail()` so they are *Red*.

- A change that affects both improving tests and worsening tests is classified as *Yellow*. For example, change 4 (**AF** for `A.y`) affects both `testPassFail()` and `testFailPass()` and therefore is *Yellow*.

Intuitively, *Red* changes are most likely to be the reason for a test failure, followed by *Yellow* changes, while *Green* changes can never be failure-inducing. How to associate colors with changes becomes less obvious when changes also affect tests that have the same outcome in both program versions. Section 3 defines a number of classifiers that follow different strategies. For two of these change classifiers (R_s/G_r, R_s/G_s), only changes 8, 9 and 10 are colored *Red*, exactly the failure-inducing changes cited earlier for this example.

3. DEFINITIONS

In this section, we define criteria for change classification and present several change classifiers based on these criteria. We implicitly make the usual assumptions [10] that program execution is deterministic and that the library code and execution environment (e.g., JVM) remain unchanged.

Our classification of tests is based on the *JUnit* test result model in which a test can *pass*, *fail* (i.e., an assertion failure) or *crash* (i.e., an *unexpected* exception is caught by the *JUnit* runtime[3]). Definition 3.1 below formalizes this test result model[4] and introduces an ordering in which passing tests are preferred over failing tests, and failing tests are preferred over crashing tests.

DEFINITION 3.1 (TEST RESULT MODEL). *Let* $\mathcal{R}$= { PASS, FAIL, CRASH} *be the set of all test results. Furthermore, we define the following ordering on test results:*

$$\text{CRASH} < \text{FAIL} < \text{PASS}$$

For a given test T, we will use $R_{orig}(T)$ and $R_{edit}(T)$ to represent the result of test T in the original program and the edited program, respectively, where $R_{orig}(T), R_{edit}(T) \in \mathcal{R}$. Definition 3.2 below uses this notation to classify tests as worsening or improving.

DEFINITION 3.2 (TEST CLASSIFICATION). *Let* $\mathcal{T}$ *be the set of all tests. Then the sets WT and IT of worsening tests and improving tests, respectively, are defined as follows:*

$$\begin{array}{lcl} WT & = & \{\, T \in \mathcal{T} \mid R_{orig}(T) > R_{edit}(T) \,\} \\ IT & = & \{\, T \in \mathcal{T} \mid R_{edit}(T) > R_{orig}(T) \,\} \end{array}$$

In the definitions below, we will use the notation $AT(A)$ to represent the tests in $\mathcal{T}$ affected by atomic change $A \in \mathcal{A}$ (i.e., the set of all atomic changes between two versions) and $AC(T)$ to represent the atomic changes affecting a given test $T \in \mathcal{T}$. Definition 3.3 defines auxiliary change sets *Worsening*, *Improving*, *SomeFailFail*, *SomePassPass*, and *OnlyPassPass*. *Worsening* and *Improving* are the sets of changes that affect at least one worsening test, or at least one improving test, respectively. *SomeFailFail* and *SomePassPass* are the sets of changes that affect at least one test that crashes/fails or passes in both versions, respectively. Finally, *OnlyPassPass* is the set of changes that only affect tests that pass in both versions.

DEFINITION 3.3 (CHANGE INFLUENCE).

$$\begin{array}{ll} \textit{Worsening} & = \{ A \mid A \in \mathcal{A},\, WT \cap AT(A) \neq \emptyset \} \\ \textit{Improving} & = \{ A \mid A \in \mathcal{A},\, IT \cap AT(A) \neq \emptyset \} \\ \textit{SomeFailFail} & = \{ A \mid \exists T \in AT(A), \\ & \quad R_{orig}(T) = R_{edit}(T) \in \{\, \text{FAIL}, \text{CRASH} \,\} \} \\ \textit{SomePassPass} & = \{ A \mid \exists T \in AT(A), \\ & \quad R_{orig}(T) = R_{edit}(T) = \text{PASS} \} \\ \textit{OnlyPassPass} & = \{ A \mid \forall T \in AT(A), \\ & \quad R_{orig}(T) = R_{edit}(T) = \text{PASS} \} \end{array}$$

We can now classify changes as *Red*, *Yellow*, or *Green*. Intuitively, our goal is to classify changes such that *Red* changes are highly likely to be the reason for test failures, *Yellow* changes are possibly problematic, and *Green* changes are correlated with successful tests. There are several ways in which one could design such a classifier, and it was not clear to us *a priori* which approach would work best in practice. As we wanted to explore the potential of change classification, our approach was to define five different classifiers that each partition the set of changes into *Red*, *Yellow*, and *Green* subsets in slightly different ways. In Section 4 we will present a comparative evaluation of these different classifiers in two case studies.

The first classifier is called *simple* and relies *only* on test results in the edited program. A change is classified *Red* if it only affects failing or crashing tests, *Green* if it only affects passing tests, and *Yellow* otherwise. To define the remaining four classifiers, we use a *relaxed* and a *strict* criterion based on the *development of test results* for the two versions for each color, as shown in Table 1. We will refer to these criteria as R_r, R_s, G_r and G_s, where the capital letter represents the color, and the subscript represents the criterion used, where *r* indicates *relaxed* and *s strict*. Tests that are new or

[3] Note that this situation is distinct from the one where the *expected* outcome of a test is an exception, in which case the test itself should catch the exception.

[4] Our approach can easily be adapted to accommodate other test result models with, for example, a single error state, multiple fine-grained error states, or a model in which FAIL is a worse result than CRASH.

	Criteria	
Coloring	**relaxed**	**strict**
Red	R_r: $(A \notin Improving \wedge A \in Worsening)$	R_s: $(A \notin Improving \wedge A \in Worsening \wedge A \notin SomePassPass)$
Green	G_r: $A \in OnlyPassPass \vee (A \in Improving \wedge A \notin Worsening)$	G_s: $A \in OnlyPassPass \vee (A \in Improving \wedge A \notin Worsening \wedge A \notin SomeFailFail)$
Yellow	$A \notin Red, A \notin Green, AT(A) \neq \emptyset$	

Table 1: Definitions of four methods for classifying atomic changes into *Red*, *Yellow*, **and** *Green* **changes.**

that have been deleted in the edited program have no effect on the classifiers built from these criteria, as they do not correlate with improved or degraded test results. The classifiers have been defined in such a way that the set of changes colored *Red* by the R_s classifiers is a subset of those colored *Red* by the R_r classifiers. Similarly, the set of changes colored *Green* by the G_s classifiers is a subset of those colored *Green* by the G_r classifiers.

Intuitively, the G_r criterion marks as *Green* any change that affects improving tests but not worsening tests, as well as any change that only contributes to tests that succeed in both versions of the program. While this is a reasonable criterion, it may have the somewhat counterintuitive effect that a *Green* change may affect a test that fails in the edited version of the program. In the example in Figure 1, change 2 affects both `testFailPass()`, an improving test, and `testFailFail()`; it will be colored *Green* by the G_r criterion. The G_s criterion eliminates such potentially confusing effects by requiring that all *Green* changes must only affect tests that succeed in the edited program, causing change 2 to be colored *Yellow*. Note that both the G_r and the G_s criteria have the desirable property that changes classified as *Green* are never failure-inducing, since they never affect any worsening test.

The difference between R_r and R_s is similar. The R_r criterion marks as *Red* any change that affects worsening tests but not improving tests. This is reasonable, but it may have the counterintuitive effect that a change that affects a test succeeding in both versions of the program may still be *Red* (e.g., change 1 in our example). The R_s criterion further restricts *Red* changes to affect only tests that FAIL or CRASH in the edited program. Any changes that are colored neither *Red* nor *Green* are classified as *Yellow* if they affect some tests. As we can apply these two criteria for *Red* and *Green* independently, we obtain four classifiers by combining them. We will refer to these classifiers as R_r/G_r, R_s/G_r, R_r/G_s, and R_s/G_s.

Note that there is an asymmetry in the four non-*simple* change classifiers. A change that affects only tests that pass in both versions is always classified as *Green*, whereas a change that affects only tests that FAIL in both versions (or CRASH in both versions) is always classified as *Yellow*. To motivate this decision, recall that the purpose of our change classification is to reveal failure-inducing changes. A change that only affects passing tests by definition is not failure-inducing (for the current test suite) and is therefore classified as *Green*. In contrast, if a change A affects a test that fails in both versions, the failure in the edited version may reflect the same problem as before, or it may now be due to A; therefore, *Yellow* seems a more appropriate choice than *Red*.

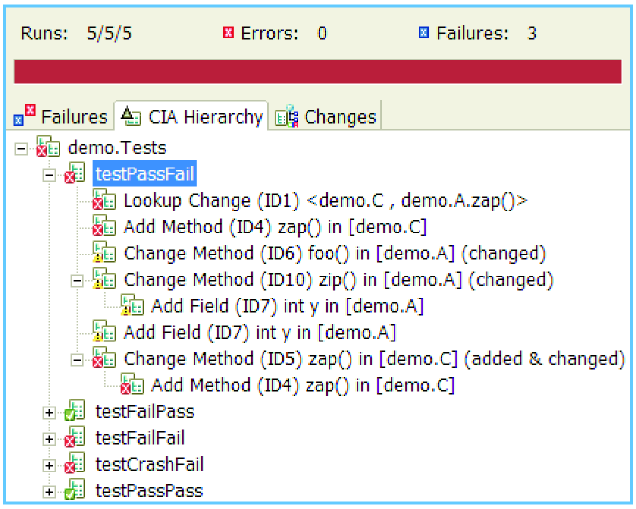
Some changes do not affect any tests. We classify a change A as *Gray*, if it affects no tests (i.e., $AT(A) = \emptyset$). This is a coverage issue rather than a debugging issue, as it indicates that the test suite should be expanded to cover *Gray* changes as well. Table 2 shows how the changes of the example of Figure 2 are classified according to our five classifiers.

Change	*simple*	R_r/G_r	R_s/G_r	R_r/G_s	R_s/G_s
1	*Yellow*	*Red*	*Yellow*	*Red*	*Yellow*
2	*Yellow*	*Green*	*Green*	*Yellow*	*Yellow*
3	*Green*	*Green*	*Green*	*Green*	*Green*
4	*Yellow*	*Yellow*	*Yellow*	*Yellow*	*Yellow*
5	*Yellow*	*Yellow*	*Yellow*	*Yellow*	*Yellow*
6	*Gray*	*Gray*	*Gray*	*Gray*	*Gray*
7	*Gray*	*Gray*	*Gray*	*Gray*	*Gray*
8	*Red*	*Red*	*Red*	*Red*	*Red*
9	*Red*	*Red*	*Red*	*Red*	*Red*
10	*Red*	*Red*	*Red*	*Red*	*Red*
11	*Gray*	*Gray*	*Gray*	*Gray*	*Gray*
12	*Red*	*Green*	*Green*	*Green*	*Green*

Table 2: Classification of the atomic changes of Figure 2 according to the *simple* **classifier and the 4 composite classifiers based on the criteria defined in Table 1.**

4. IMPLEMENTATION AND EVALUATION

To evaluate our change classifiers we created the tool *JUnit/CIA*, implemented as an Eclipse plug-in that builds on the analysis component of the *Chianti* tool we previously developed [24]. *JUnit/CIA* uses the version of the program that is currently in the Eclipse workspace as the *edited version*, and either uses another existing project as the *original version* or retrieves a previous version from the local history that corresponds to the last time the test suite was executed. (The local history is a local RCS repository maintained by Eclipse that records all textual changes.) Dynamic call graphs for the tests are obtained by monitoring their execution using the JVMPI profiling interface.



Runs: 5/5/5 Errors: 0 Failures: 3

demo.Tests
testPassFail
Lookup Change (ID1) <demo.C , demo.A.zap()>
Add Method (ID4) zap() in [demo.C]
Change Method (ID6) foo() in [demo.A] (changed)
Change Method (ID10) zip() in [demo.A] (changed)
Add Field (ID7) int y in [demo.A]
Add Field (ID7) int y in [demo.A]
Change Method (ID5) zap() in [demo.C] (added & changed)
Add Method (ID4) zap() in [demo.C]
testFailPass
testFailFail
testCrashFail
testPassPass

Figure 5: *JUnit/CIA* **hierarchy view**

The user-interface of *JUnit/CIA* extends that of the *JUnit* Eclipse component as follows: (i) in the *CIA* hierarchy view, affecting changes are shown in a tree-view underneath each test, where expanding the tree reveals prerequisite changes (see Figure 5), and (ii) an additional view shows all the changes organized by category (i.e., **AM**, **CM**, etc.). In each of these views, colored icons are associated with changes to indicate if they are *Red*, *Yellow*, *Green*, or *Gray*, and double-clicking on a change causes a standard Eclipse compare view of the associated original and edited code to appear.

In order to improve performance, we implemented a filtering mechanism that allows users to avoid tracing of methods in the standard libraries. Although, by assumption, such methods do not contain any changes, they may execute virtual method calls that dispatch to methods in user code (i.e., call-backs), and such dispatch operations may exhibit changed behavior when overridden library methods are added, deleted, or changed. We conservatively approximate the behavior of call-backs using an approach similar to that of [34].

Definition of failure-inducing changes. To assess the quality of our results we need those changes that are actually failure-inducing. Given a worsening test[5], we can selectively undo a subset of its affecting changes, and observe whether or not the test outcome on the resulting intermediate program is worsening, with respect to the original version outcome. If the test is not worsening (on the intermediate version), then that subset contains *failure-inducing* changes. For our case studies, we manually derived the failure-inducing change sets for each application, making a best effort to obtain as small a subset as possible. Ideally, our classifiers should color exactly these changes *Red*.

4.1 Case Study 1: Student Projects

During our first case study with student projects, we encountered several situations where tests did not terminate. To handle such cases, we implemented a time-out mechanism where the execution of a test is aborted after a specified number of seconds. (In our experiments, we used a time-out of 10 seconds.) In such cases, we used the dynamic call graph obtained by executing the program up to that point, and consider the test result to be Crash. We extended the standard *JUnit* launch configuration to allow users to specify this time-out option.

Overall, we analyzed source code from 40 small student projects of an undergraduate programming course at the University of Passau. In this course, students implemented Dinic's Maximum Flow algorithm using a predefined set of mandatory interfaces. The students were provided with a set of public *black box* tests that had to be successfully executed in order for students to pass the course. We also defined an additional *secret* test suite, whose existence was known to the students, although no details of these tests were available. Although the students had to agree that their code could be used for research purposes, they did not know that their data would be used to evaluate change classifiers. Course management was provided using the web-based *Praktomat* system [39]. Students frequently submitted their solutions to Praktomat, which then automatically compiled them and ran the tests. Praktomat automatically saves all submitted versions in a database, so that these versions were available to us for this case study.

Analyzed code base. Some minor postprocessing of the student code was needed to make it suitable for our experiments. As Praktomat uses black box testing, the public tests were coarse-grained regression tests for `DejaGNU`, an open-source black box regression-testing framework[6]. Our postprocessing consisted of writing equivalent *JUnit* tests with assertions based on the mandatory interfaces, and adding fine-grained unit tests. In a few cases, several interpretations of the mandatory interfaces existed (e.g., node numbering in the graph could start at 0, or at 1), and we rewrote the tests for specific student solutions to uniformly use the same approach. We also commented out debugging output in a few cases for performance reasons. None of these changes affected the semantics of the submitted code in fundamental ways.

number of version pairs	
written by students	1175
that contain meaningful changes	556
with associated worsening tests	110
with identifiable failure-inducing changes	98
where versions pairs differ by >1 change	61

Table 3: Selection of version pairs from the student data.

On average, each of the final, graded solutions consisted of 950 lines of commented Java source code. We analyzed a total of 1175 version pairs written by 40 students. Of these 1175 version pairs, 556 contained meaningful changes[7], and 110 of these 556 version pairs had associated worsening tests. For 98 of these 110 version pairs, we could manually identify the *failure-inducing changes*. In the remaining 12 cases we were unable to determine the failure-inducing changes due to the size of the edit or non-deterministic test behavior. Since we are interested in techniques for automatically determining failure-inducing changes, we need version pairs that differ by more than one change (otherwise, the reason for the failure is obvious). Eliminating the version pairs that differ by one change resulted in a final set of 61 version pairs (out of the 98) that we used as the basis for evaluating the 5 change classifiers presented in Section 3. The process of selecting version pairs is illustrated by Table 3.

Per-Version-Pair Evaluation. The 61 version pairs contained a total of 1295 atomic changes of which 894 were *Gray*, leaving 401 non-*Gray* changes. Table 4 shows how the different classifiers associate colors with these changes. From left to right, the columns of the table indicate the total number of changes classified as *Red*, *Yellow*, *Green*, and *Gray* respectively. For example, the R_r/G_r classifier finds 138 *Red*, 126 *Yellow*, 137 *Green*, and 894 *Gray* changes.

To determine classifier quality, we manually identified the failure-inducing changes for each of the 61 version pairs. Then, we calculated *recall* and *precision* for each classifier, both for *Red* changes and for *Green* changes. These are core metrics from information retrieval theory stating the percentage of desired results retrieved, and the percentage of correctly retrieved items among all retrieved items, respectively. Due to the way in which the classifiers are defined, the choice of the criterion to classify *Green* changes has no effect on recall and precision for *Red*. Thus, we will discuss classifier results for *Red* changes independently of the *Green* criterion, and *vice versa*.

For *Red* changes, *recall* is the fraction of failure-inducing changes colored *Red*, and *precision* is the ratio of actual failure-inducing *Red* changes to all the *Red* changes. It is desirable to minimize the number of *false positives* or spurious failure-inducing changes reported. The percentage of false positives equals $1 - precision$ (of *Red* changes). Similarly, we seek to minimize the number of failure-inducing changes not colored *Red*, that is *false negatives*. The percentage of false negatives equals $1 - recall$ (of

[5] Pass to Fail, Pass to Crash, or Fail to Crash

[6] See `www.gnu.org/software/dejagnu/`.

[7] Our analysis considers two versions the same if they differ only in layout or comments. The relatively high number of versions without changes is due to coding style requirements for the course, which were addressed by the students late in their implementations.

Classifier	*#Red*	*#Yellow*	*#Green*	*#Gray*
R_r/G_r	138	126	137	894
R_s/G_r	77	187	137	894
R_r/G_s	138	200	63	894
R_s/G_s	77	261	63	894
simple	119	238	44	894

Table 4: Coloring of changes according to the 5 classifiers (cumulative statistics over 61 version pairs).

Classifier	Total pairs	*recall Red*	*false Neg.*	*prec. Red*	*false Pos.*	F_1
R_r/*	39	0.91	0.09	0.63	0.37	0.68
R_s/*	22	0.78	0.22	0.66	0.34	0.64
simple	26	0.66	0.34	0.44	0.56	0.49

Table 5: Recall, false negative rate, precision, false positive rate, and F_1 measure for R_r/*, R_s/*, and *simple* classifiers averaged over their type (ii) and (iii) colorings of the 61 version pairs.

Red changes).

When examining the effectiveness of *Red* changes to identify the failure-inducing changes, we distinguish the following three scenarios: (i) colorings in which none of the affecting changes of a test are colored *Red*, (ii) colorings in which no failure-inducing changes are colored *Red*, but where some other affecting changes are colored *Red*, thus providing information that is misleading, and (iii) colorings in which the *Red* changes included at least some of the failure-inducing changes (i.e., colorings where *Red* changes might help a programmer focus on those changes that are failure-inducing).

Table 5 shows how effectively the *Red* changes can identify failure-inducing changes. The table shows, for each of the R_r/*, R_s/*, and *simple* classifiers, the average rates of recall, precision, false positives and false negatives.[8] Also shown in the table is the average F_1 measure [36], which corresponds to the harmonic mean of recall and precision, and which can be used as a combined metric to compare classifier quality.[9] We chose the harmonic mean of recall and precision in order to minimize the number of false positives and false negatives, giving equal emphasis to them both. If a classifier results only in type (i) colorings for a given version pair, then we do not include that version pair in our averages. Thus, the second column in Table 5 reports the number of version pairs used to calculate the values in the table for each classifier. Note that ideally we would like a classifier with a value of 1.0 for both recall and precision, thus eliminating these false reports completely.

It is easy to see that the *simple* classifier can be dismissed, because it has the lowest F_1 value. This means that it is not as good at avoiding both false negatives and false positives as either the R_r/* or R_s/* classifiers. On average, the R_r/* classifiers yield marginally higher F_1 values than the R_s/* classifiers (0.68 vs 0.64). However, they also produce distinguishing colorings (i.e., types (ii) and (iii)) over significantly more version pairs in the data (i.e., 39 vs 22). Therefore, we choose the R_r/* classifiers over the R_s/* as best for this data.

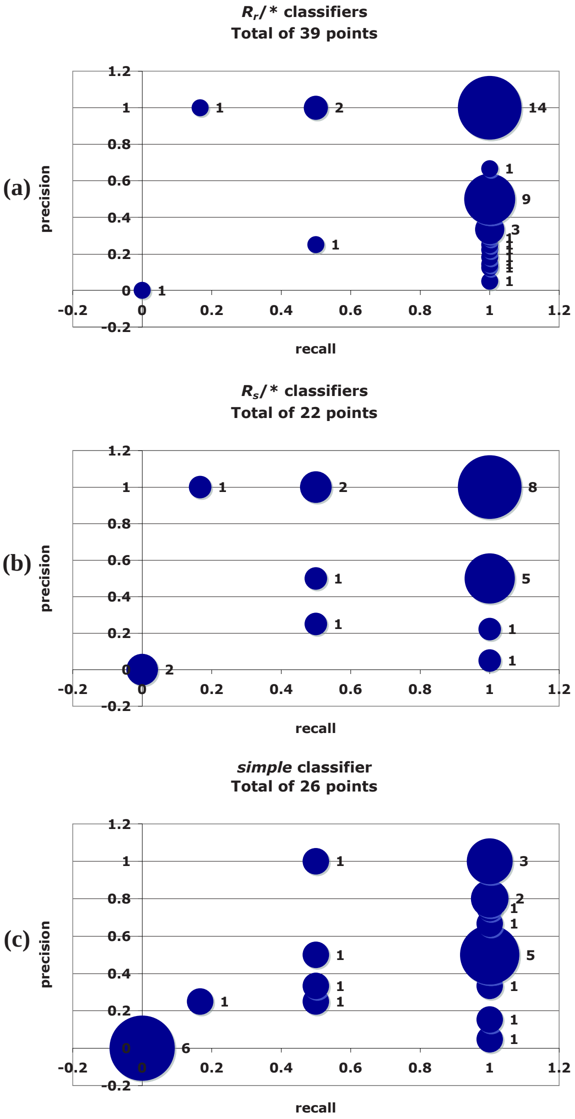


Figure 6: Scatter plots for recall and precision values obtained per version pair for (a) R_r/*, (b) R_s/*, and (c) *simple* classifiers.

Figure 6 shows, for each of the R_r/*, R_s/*, and *simple* classifiers, a scatter plot of the pairs of recall and precision values obtained for each version pair presented in Table 5. In these charts, the size of each bubble reflects the number of same-valued pairs it represents, which appears explicitly as a number next to the bubble. Bubbles close to the upper right corner of the graph represent the most desirable values (i.e., approximately 1.0 for both recall and precision). Bubbles centered at the origin represent the type (ii) colorings, with 0 for both recall and precision, because no failure-inducing changes are colored *Red*, although there are some *Red* changes. Thus, these bubbles represent *misleading* colorings, because some non-failure-inducing changes are *Red*, and all failure-inducing changes are *Yellow*. Clearly, a good classifier should produce as few as possible misleading colorings. Over our 61 version pairs, recall that the R_r/* classifiers produce one such coloring, R_s/*

[8] For an overview of the non-aggregated data, the reader is referred to [32].

[9] Values for version pairs with zero for both recall and precision, (i.e., type (ii) colorings for a particular classifier) were included in the average by using an F_1 value of zero. We included these cases to be conservative; such cases demonstrate bad performance for the corresponding classifier and the worst possible value for F_1 is zero. There was 1 such version pair for R_r/*, 2 for R_s/*, and 6 for *simple*.

two and *simple* six; thus, considering this quality measure also, the R_r/* classifiers are the best.

For *Green* changes, recall is the fraction of all non-failure inducing changes colored *Green* and precision is the ratio of actual non-failure inducing changes to all *Green* changes. In selecting the preferred *Green* criterion, we only consider recall for the *Green* changes, because their precision will be 1.0 since *Green* changes never affect worsening tests (by definition). Over the 61 version pairs, the */G_r classifiers produce a recall of 0.19 for *Green* changes, versus 0.15 for the */G_s classifiers, meaning that the former are more successful at classifying non-failure-inducing changes as *Green*. Consequently, the */G_r classifiers are clearly preferable to the */G_s ones. In summary, for this case study, it is clear that the R_r/G_r classifier produces the best results.

Per-Test Evaluation. As a final step in this case study, we measured how often change classification may help the programmer find the failure-inducing changes for a given test failure. For this "per test" view, we examine 444 worsening tests in the 61 version pairs under consideration that have 2 or more affecting changes. The baseline for comparison is the uncolored set of affecting changes as calculated by *Chianti*. For 211 of the 444 worsening tests, we calculated that the R_r/* classifiers colored all the failure-inducing changes *Red*, and some of the other affecting changes *Yellow*. This means that the R_r/* classifiers were successful at focusing programmer attention in 47.5% (211/444) of the tests. This property was evidenced in only 25.5% (113/444) of the worsening tests using the R_s/* classifiers and 15.1% (67/444) for the *simple* classifier.

Conclusions. The accuracy of the R_r/* classifiers in identifying the failure-inducing changes confirms that they performed the best in our student projects case study. Thus, this study selects the R_r/G_r classifier as best.

4.2 Case Study 2: Daikon

Daikon [8] is a system for discovering likely invariants in software systems using dynamic analysis. We extracted several versions of Daikon from the CVS repository, but (unfortunately for our purpose) could not find any worsening unit tests. This illustrates a common problem in obtaining evaluation data for our method. In general changes are only committed if all tests succeed, (i.e., there are no worsening tests in repositories). However, we noticed that several unit tests changed between the Daikon versions *Daikon/2002-11-11* and *Daikon/2002-11-19*, and reusing the old tests with the edited version produced 2 test failures. In the experiments discussed below, we treat these test failures as worsening tests. For the Daikon version pair under consideration, a total of 61 tests were defined, of which 40 were affected by the edit (there were also 7 new tests and 3 deleted tests). The two versions differed significantly, as a total of 6093 atomic changes were reported by our tool.

The first test, `testXor`, was affected by 35 atomic changes. Manual inspection of the code revealed that two **CM** changes to methods `daikon.diff.XorVisitor.shouldAddInv1()` and `daikon.diff.XorVisitor.shouldAddInv2()` were responsible for the test's failure. The R_r/* classifiers failed to focus on these changes since they classified all 35 affecting changes as *Red*, because there were no improving tests in this experiment. Both R_s/* classifiers correctly identified the 2 failure-inducing changes as *Red*, as well as 2 of 33 remaining changes, with the rest classified as *Yellow*. In other words, the R_s/* classifiers were very successful at correctly focusing the programmer's attention on only 4 out of 35 affecting changes in this case, including the appropriate ones.

The second test, `testMinus`, produced a similar result. This test was affected by 34 changes, and we manually identified the failure-inducing change to be a **CM** change to method `daikon.diff.Diff.shouldAdd()`. Again, the R_r/* classifiers were not useful because they classified all 34 changes as *Red*. The R_s/* classifiers were very effective by classifying the failure-inducing change as *Red*, only two other changes as *Red*, and the 31 remaining changes as *Yellow*. Thus, the programmer's attention was focused on only 3 of 34 changes, including the appropriate one.

The tests for these two Daikon versions were either worsening (i.e., PASS to FAIL) or successful in both versions (i.e., PASS to PASS). Because of this restricted set of test outcomes, the *simple* classifier and the R_s/* classifiers produce the same coloring of the changes. If there had been tests that failed in both versions (i.e., FAIL to FAIL), then the *simple* classifier would have been less successful at focusing the programmer's attention on failure-inducing changes than the R_s/* classifier. These limited test outcomes seem coincidental, and may be the result of our constructed tests; therefore, we prefer the R_s/* classifiers for the Daikon data.

Of the 6093 changes separating the two Daikon versions, 5715 were classified as *Gray* due to the low coverage of the Daikon unit test suite. Of the remaining 378 changes, 338 were *Green*, 33 *Yellow*, and only 7 *Red* using the R_s/G_r classifier. In this case study, our approach reduced the number of changes to be examined from 6093 to 35 (or 34) affecting changes for each worsening test, and then further reduced the number to 4 (or 3) using the *Red* changes obtained from the R_s/* classifiers. Note that programmers do not know up front if a change is covered by a test suite, (i.e., classifying a change as *Gray* is also necessary).

Conclusions. The R_s/* classifiers outperformed the R_r/* classifiers on the Daikon version pair, by focusing programmer attention on the failure-inducing change(s) by coloring them *Red*. Thus, this study selects the R_s/G_r classifier as best.

4.3 Assessment

With our current untuned research implementation of *JUnit/CIA*, constructing dynamic call graphs slows down the execution of tests by more than a factor of 10. For the student project case study this was insignificant, but for the Daikon case study, the timings were unacceptable for interactive use. In our experiments with Daikon, constructing the dynamic call graphs for all unit tests for a given version takes about 4 minutes on average, and computing and classifying the atomic changes takes less than 2 minutes. The performance problem is primarily due to the annotation of static receiver types on the constructed dynamic call graphs, collected using a JVMPI agent. Unfortunately, type information is not available through JVMPI; we will explore other approaches for dealing with this problem. Currently, to address this performance issue, we suggest a scenario where programmers run their tests normally, until they encounter a worsening test. Only then do they rerun the tests using *JUnit/CIA* to perform change classification.

The main outcome of our two case studies is a positive demonstration that change classification may focus programmer attention on parts of an edit that may be the root cause of unexpected worsening test behavior. While more extensive empirical investigation of larger programs is necessary to fully validate this claim, the success of change classification in these studies is undeniable. Currently, we plan to further validate our approach using larger programs such as the Eclipse *jdt core*.

As is common, the case studies also raise unexpected questions. For example, the two case studies select contradictory choices for the "best" classifier, but this can be explained by considering the behavior of the associated test suites. In defining our classifiers, we have assumed that the parts of the program executed by different tests will overlap, and therefore, some changes will affect more than one test. If this assumption is violated, then all the changes affecting a worsening test will be colored *Red,* offering no focus on the failure-inducing changes. Thus, the success of classification depends on some properties of the tests used.

In both case studies, the number of *Gray* changes is considerable. In the student programs case study, this is due to the fact that, initially, the students did not use the mandatory interfaces but instead implemented a solution based on their own interfaces and only adapted their solution to use the mandatory interfaces later. Consequently, our tests failed to execute relevant portions of their code until the mandatory interfaces were adopted, resulting in many *Gray* changes in the earlier stages of the implementations. The students also tended to provide considerably more elaborate command-line interfaces than we expected, and this untested code resulted in more *Gray* changes. In the case of Daikon, the coverage of the unit test suite is fairly low.[10] Any changes to parts of Daikon not covered by the unit test suite will be colored *Gray*. Daikon also has a suite of regression tests with presumably much better coverage, but unfortunately those regression tests are not based on JUnit, and thus applying our technique would be difficult. Consequently, while our findings are promising, they would be more compelling with a test suite offering better coverage of the system.

The student projects exhibit a mixture of improving and worsening tests, and the R_r/* classifiers work best here. On the other hand, in the Daikon study, there are only a few worsening tests and no improving tests. The R_r/* classifiers are hindered by the lack of improving tests, which prevents any affecting changes of a worsening test from being colored *Yellow*. Since the R_s/* classifiers do not color changes *Red* that affect tests with the same outcome in both the original and edited program, they are able to focus programmer attention on a *subset* of the changes affecting the worsening tests. Thus the R_s/* classifiers perform better on the Daikon case study.

Given these empirical results, we suggest that programmers use the R_r/G_r classifier during development when both improving and worsening tests exist. If only worsening and same-outcome tests occur, then the R_s/G_r classifier seems to be the better choice. It is possible that through experience, development organizations will be able to select the appropriate classifier for their projects.

Additional questions raised by our investigations include the following: Does the choice of classifier depend on other factors we have not yet considered, including programmer experience level, software maturity (i.e., in active development versus maintenance), etc.? Are there other interesting classifiers to investigate (e.g., using the frequency that a change affects a worsening test to obtain a statistically-based change coloring, reminiscent of the statement coloring in [13])? Are there properties of the test suite which can suggest the appropriate classifier to use? We will explore these issues as future research.

5. RELATED WORK

Delta Debugging. In the work on *delta debugging,* the reason for a program failure is identified as a set of differences between versions [38], inputs [41], thread schedules [4], or program states [40, 5] that distinguish a succeeding program execution from a failing one. A set of failure-inducing differences is determined by repeatedly applying different subsets of the changes to the original program, and observing the outcome of executing the resulting intermediate programs. By examining the outcome of each execution (*pass, fail,* or *inconsistent*), the set of failure-inducing changes is narrowed down using efficient binary-search techniques.

Our work and delta debugging are different approaches for identifying failure-inducing changes, each with its strengths and weaknesses. Delta debugging determines whether or not a change is failure-inducing by observing the effect of its presence or absence in two program executions. Executing intermediate program versions helps narrow down the reason for a program failure but, in the worst case, a number of executions proportional to the number of changes is required. In contrast, our approach identifies reasons for failures using the results of distinct tests that execute different subsets of the changes, and requires a suite of tests with this property. The two approaches may complement each other. In principle, the use of a rich model of changes with interdependences could improve the efficiency of delta debugging by reducing the number of intermediate programs that are constructed/executed. Conversely, our method could be made more precise by executing tests on intermediate program versions, and taking their results into account.

Comparing Dynamic Data From Different Executions. Several debugging approaches rely on comparing dynamic information associated with succeeding and failing runs. Reps et al. [26] compare path profiles from different executions in order to expose incorrect Year 2000 date-related computations that give rise to the execution of different paths. Harrold et al. [11] evaluate the effectiveness of comparing path profiles (and other run-time metrics) for distinguishing successful executions from failing ones. They found a strong correlation between differences in path profiles and different execution behavior; similar findings held for their other metrics. Jones et al. [13, 12] use the colors red, yellow, and green to visualize the statements executed by failing tests only, by both succeeding and failing tests, and by passing tests only, respectively. They found this *discrete* visualization to be “not very informative, as most of the program is yellow” and also propose an *continuous* visualization where a gradual scale of color and brightness reflects both the absolute number of tests, and the relative percentages of passing and failing tests that execute a given statement. Our work differs from their discrete approach because we visualize the correlation between *changes* and their affected tests, whereas Jones et al. visualize the correlation of *statements* with test results. Our approach is likely to be more useful for locating failure-inducing changes because the number of executed changes is likely to be far smaller than the number of executed statements, and because the execution of different statements by a failing test may be due to a change in a completely different part of the program. Ruthruff et al. [27] also use a continuous color scale to indicate the contribution of cells in a spreadsheet to incorrect values. In this work, the user indicates whether or not computed values are correct, and dependences between cells are used to compute the likelihood that (the formula in) a given cell contributes to an incorrect value.

Renieris and Reiss [25] use tracing data from one faulty and several successful runs to detect failures in C programs. They build a model from the traces, calculate a difference between the models of the faulty and the successful runs and map this difference back to source code artifacts, which finally forms the report. Dallmeier et al. [6] present a technique for localizing errors by comparing sequences of method calls in passing and failing runs of a program. Their experiments indicate that comparing method call sequences is a better defect indicator than a simple coverage-based metric, such as the one by Jones et al. [13], and that comparing sequences of method calls *on the same object* is an even better predictor.

[10] For the Daikon versions under consideration in [24], we reported that the unit test suite covered 21% of the methods on average for these versions. However, this number is skewed by the fact that certain Daikon components have reasonable coverage (e.g., for the `util.MDE` component we find an average coverage ratio of 47%), whereas other components (e.g., the `jtb` component) have virtually no coverage.

Statistical Techniques. Some researchers use statistics to calculate the likelihood that a specific predicate is related to a fault. Liblit et al. [17, 18] present statistical analyses in which information is gathered about the number of times that certain predicates are executed by deployed applications, in order to detect predicates whose outcome correlates with a crash. A low sampling frequency is used to ensure low run-time overhead, so a large number of samples is needed to obtain meaningful data. A number of strategies is presented that allow one to quickly rule out certain predicates as being related to failures. Liu et al. [19] propose a statistical model-based approach to localize bugs and define the "evaluation bias" of a predicate, which measures the probability of a predicate being "true" in one execution. Then, the evaluation patterns in correct and incorrect runs are compared to identify those predicates that are likely to be bug-relevant. A comparison of their model with Liblit's method [18] and Cleve and Zeller's method [5] shows that they can localize more bugs (68/130 in the Siemens suite) in certain contexts. While we do not use statistical methods to classify changes yet, investigating new classifiers based on such methods might be a fruitful area for future work.

Fault Localization Techniques. A program slice [37, 33] w.r.t. an incorrect value contains all statements that may have contributed to that value, and will generally include the statement(s) that contain the error. Slices may become very large, and techniques such as *dicing* [20] have been proposed, where a slice w.r.t. an erroneous value is intersected with a slice w.r.t. a correct value. DeMillo et. al. [7] define a *critical slice* w.r.t. a failing test t to contain all "critical" statements that, when omitted, cause program execution to reach a designated failure statement with different values for referenced variables. Gupta et al. [9] propose an approach that integrates delta debugging with program slicing to narrow down the search for faulty code. First, delta debugging is used to identify a minimal failure-inducing input, and a forward dynamic slice is computed from this input. Then, they obtain a backward dynamic slice with respect to the erroneous output, and the intersection of these two slices may potentially contain the faulty code.

Our approach and program slicing can both be used for finding faults, but there are two significant differences. Slicing is a fine-grained analysis at the statement level that can be used to inspect a failing program to help locate the cause of the failure. Our work focuses on failures that are due to the application of a set of changes, and our analysis is at the method level.

Change Impact Analysis. We previously presented a conceptual framework [28] for change impact analysis, and its expansion to the full Java language with empirical validation [24]. We also have developed a tool for building intermediate program versions by applying a subset of affecting changes to the original program [3, 23]; this tool was used to identify failure-inducing changes in the second case study. In this paper, we use change classification to identify a *a subset of the changes* that are responsible for a given test's failure. Other research on impact analysis aims at finding *program constructs* potentially affected by changes. These analyses are based on static analysis [2, 16, 14, 35], dynamic analysis [15] or, like our analysis, on a combination of the two [21]. Recent work on change impact analysis includes the *PathImpact* algorithm by Law and Rothermel [15], where dynamic call information is used to determine the procedures potentially impacted by a change to a procedure p, and the *CoverageImpact* technique by Orso et al. [21], which combines the use of a forward static slice [33] w.r.t. a changed program entity (i.e., a basic block or method) with execution data obtained from instrumented applications to find affected program entities. An empirical comparison of these algorithms appears in [22].

Continuous Testing and Test Factoring. Saff and Ernst present two techniques for identifying test failures early, when reasons for these failures are easy to identify. In *continuous testing* [29, 31], tests are run whenever the CPU is idle. *Test factoring* [30] automatically derives fast unit tests from slow system-wide tests using dynamic analysis. Change classification complements these techniques by reducing the amount of time needed to fix bugs.

6. CONCLUSIONS AND FUTURE WORK

There are three main contributions of this paper. First, we presented an approach for change classification that helps programmers identify the changes responsible for test failures. As part of this approach, we proposed several change classifiers that associate the colors *Red*, *Yellow*, or *Green* with changes, according to the likelihood that they were responsible for test failures. Second, we implemented these change classification techniques in *JUnit/CIA*, an extension of the *JUnit* component of Eclipse. Third, we conducted two case studies in which we investigated whether or not change classification can be a useful tool for focusing the programmer's attention on failure-inducing changes.

Furthermore, in response to the 3 research questions posed in Section 1, we conclude that:

- In the two case studies, change classification could successfully distinguish failure-inducing changes from other changes. Specifically, in the student programs case study, programmer attention was focused on failure-inducing changes in 47.5% of the worsening tests. In the Daikon case study, programmer attention was focused very effectively on a small superset of the failure-inducing changes.

- There is no single change classifier that always works best. In the student programs case study, R_r/G_r is the classifier of choice. However, in the *Daikon* case study, R_r/G_r failed to provide any focus on the failure-inducing changes, and the R_s/G_r classifier was highly effective.

- Based on these results, and on the characteristics of the systems being analyzed we suggest that programmers use the R_r/G_r classifier during initial development, when small differences between versions exist along with a mixture of improving and worsening tests. If versions differ more significantly, and if only worsening tests occur, then the R_s/G_r classifier seems to be the better choice.

While these results are promising, it is clear that more experimentation and/or a user study are needed for a conclusive validation of the approach. Other topics for future work include an in-depth analysis of factors we have not considered so far such as programmer experience level and properties of test suites. We also plan to develop other classifiers that, for example, take into account the frequency that a change affects a worsening test.

Acknowledgements. We are grateful to Martin Robillard, Gregg Rothermel, Joe Ruthruff, Andreas Zeller, and the anonymous reviewers for their constructive comments on previous versions of this paper. We also appreciate the excellent help provided by Ophelia Chesley in the experimentation, and the advice provided by David Madigan and Vadakkedathu T. Rajan on the analysis of the student programs case study. This research was supported by NSF grant CCR-0204410 and, in part, by IBM Research.

7. REFERENCES

[1] Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[2] Bohner, S. A., and Arnold, R. S. An introduction to software change impact analysis. In *Software Change Impact Analysis*, S. A. Bohner and R. S. Arnold, Eds. IEEE Computer Society Press, 1996, pp. 1–26.

[3] Chesley, O., Ren, X., and Ryder, B. G. Crisp: A debugging tool for Java programs. In *21st IEEE International Conference on Software Maintenance (ICSM), Budapest, Hungary* (September 2005), pp. 401–410.

[4] Choi, J.-D., and Zeller, A. Isolating failure-inducing thread schedules. In *Proc. ACM SIGSOFT Int. Symp. on Softw. Testing and Analysis (ISSTA 2002)* (Rome, Italy, 2002), pp. 210–220.

[5] Cleve, H., and Zeller, A. Locating causes of program failures. In *Proc. 27th Int. Conf. on Softw. Engineering (ICSE 2005)* (St. Louis, MO, 2005).

[6] Dallmeier, V., Lindig, C., and Zeller, A. Lightweight defect localization for Java. In *Proc. 19th European Conf. on Object- Oriented Programming (ECOOP'05)* (Glasgow, Scotland, 2005).

[7] DeMillo, R. A., Pan, H., and Spafford, E. H. Critical slicing for software fault localization. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis* (New York, NY, USA, 1996), ACM Press, pp. 121–134.

[8] Ernst, M. D. *Dynamically discovering likely program invariants*. PhD thesis, University of Washington, 2000.

[9] Gupta, N., He, H., Zhang, X., and Gupta, R. Locating faulty code using failure-inducing chops. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)* (Long Beach, California, November 2005), pp. 263–272.

[10] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A. Regression test selection for Java software. In *Proc. of the ACM SIGPLAN Conf. on Object Oriented Programming Languages and Systems (OOPSLA'01)* (October 2001), pp. 312–326.

[11] Harrold, M. J., Rothermel, G., Wu, R., and Yi, L. An empirical investigation of program spectra. In *Proc. of the ACM SIGPLAN Workshop on Program Analysis for Softw. Tools and Engineering (PASTE'98)* (Montreal, Canada, 1998), pp. 83–90.

[12] Jones, J. A., and Harrold, M. J. Empirical evaluation of the Tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)* (Long Beach, California, November 2005), pp. 273–282.

[13] Jones, J. A., Harrold, M. J., and Stasko, J. Visualization of test information to assist fault localization. In *Proc. Int. Conf. on Softw. Engineering (ICSE'02)* (Orlando, FL, 2002), pp. 467–477.

[14] Kung, D. C., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., and Chen, C. Change impact identification in object oriented software maintenance. In *Proc. of the Int. Conf. on Softw. Maintenance* (1994), pp. 202–211.

[15] Law, J., and Rothermel, G. Whole program path-based dynamic impact analysis. In *Proc. of the Int. Conf. on Softw. Engineering* (2003), pp. 308–318.

[16] Lee, M., Offutt, A. J., and Alexander, R. T. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proc. 34th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS USA'00)* (Santa Barbara, CA, 2000).

[17] Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I. Bug isolation via remote program sampling. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'03)* (San Diego, CA, 2003), pp. 141–154.

[18] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. Scalable statistical bug isolation. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'05)* (Chicago, IL, 2005).

[19] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. Sober: statistical model-based bug localization. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2005), ACM Press, pp. 286–295.

[20] Lyle, J., and Weiser, M. Automatic bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications* (Beijing (Peking), China, 1987), pp. 877–883.

[21] Orso, A., Apiwattanapong, T., and Harrold, M. J. Leveraging field data for impact analysis and regression testing. In *Proc. of European Softw. Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Softw. Engineering (ESEC/FSE'03)* (Helsinki, Finland, September 2003).

[22] Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., and Harrold, M. J. An empirical comparison of dynamic impact analysis algorithms. In *Proc. of the Int. Conf. on Softw. Engineering (ICSE'04)* (Edinburgh, Scotland, 2004), pp. 491–500.

[23] Ren, X., Chesley, O., and Ryder, B. G. Crisp: A debugging tool for Java programs. *IEEE Transactions on Software Engineering* (April 2006). In press.

[24] Ren, X., Shah, F., Tip, F., Ryder, B. G., and Chesley, O. Chianti: a tool for change impact analysis of Java programs. In *Proc. of the ACM SIGPLAN Conf. on Object Oriented Programming Languages and Systems (OOPSLA'04)* (Vancouver, Canada, October 2004), pp. 432–448.

[25] Renieris, M., and Reiss, S. Fault localization with nearest neighbor queries. In *In Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (Montreal, Quebec, Canada, October 2003), pp. 30–39.

[26] Reps, T., Ball, T., Das, M., and Larus, J. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proc. of the 6th European Softw. Conf. (ESEC/FSE'97)* (1997), pp. 432–449. Springer-Verlag LNCS Vol. 1013.

[27] Ruthruff, J. R., Burnett, M., and Rothermel, G. An empirical study of fault localization for end-user programmers. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM Press, pp. 352–361.

[28] Ryder, B. G., and Tip, F. Change impact for object oriented programs. In *Proc. of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Softw. Tools and Engineering (PASTE'01)* (June 2001).

[29] Saff, D., and Ernst, M. D. Reducing wasted development time via continuous testing. In *Fourteenth Int. Symp. on Softw. Reliability Engineering* (Denver, CO, November 17–20, 2003), pp. 281–292.

[30] Saff, D., and Ernst, M. D. Automatic mock object creation for test factoring. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Softw. Tools and Engineering (PASTE'04)* (Washington, DC, USA, June 7–8, 2004), pp. 49–51.

[31] Saff, D., and Ernst, M. D. Continuous testing in eclipse. In *Proc. of the 26th Int. Conf. on Softw. Engineering (ICSE'05)* (St. Louis, MO, USA, May 2005).

[32] Stoerzer, M., Ryder, B. G., Ren, X., and Tip, F. Finding failure-inducing changes using change classification. Tech. Rep. DCS-TR-582, Rutgers University Department of Computer Science, September 2005.

[33] Tip, F. A survey of program slicing techniques. *J. of Programming Languages 3*, 3 (1995), 121–189.

[34] Tip, F., Sweeney, P. F., Laffra, C., Eisma, A., and Streeter, D. Practical extraction techniques for Java. *ACM Trans. on Programming Languages and Systems 24*, 6 (2002), 625–666.

[35] Tonella, P. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Trans. on Softw. Engineering 29*, 6 (2003), 495–509.

[36] van Rijsbergen, C. *Information Retrieval*. Butterworths, London, 1979.

[37] Weiser, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[38] Zeller, A. Yesterday my program worked. Today, it does not. Why? In *Proc. of the 7th European Softw. Engineering Conf./7th ACM SIGSOFT Symp. on the Foundations of Softw. Engineering (ESEC/FSE'99)* (Toulouse, France, 1999), pp. 253–267.

[39] Zeller, A. Making students read and review code. In *ITiCSE '00: Proc. of the 5th annual SIGCSE/SIGCUE ITiCSE Conf. on Innovation and technology in computer science education* (2000), ACM Press, pp. 89–92.

[40] Zeller, A. Isolating cause-effect chains from computer programs. In *Proc. ACM SIGSOFT 10th Int. Symp. on the Foundations of Softw. Engineering (FSE 2002)* (Charleston, SC, 2002), pp. 1–10.

[41] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Trans. on Softw. Eng. 28*, 2 (2002), 183–200.