Change Classification and its Applications to Program Development and Testing

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

Barbara G. Ryder and Xiaoxia Ren
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

During program development, testing and code editing are interleaved activities. When tests unexpectedly fail, the changes that caused the failure are not always easy to find. We present several analyses that automatically classify changes as *Red*, *Yellow*, or *Green*, indicating the likelihood that they contributed to a test's failure. We implemented these techniques as an extension of the *JUnit* testing framework, and evaluated their effectiveness in two case studies. Our results indicate that change classification can effectively focus programmer attention on failure-inducing changes, improving on current manual searching/debugging techniques. Change classification can also determine untested changes, to inform programmers that additional tests are needed. Furthermore, change classification can determine those changes that can be committed safely to a version control repository without breaking any tests, even if a developer's local workspace contains failing tests. Early adoption of edited code avoids inefficient parallel implementations of the same functionality and possible conflicts when merging independent changes later in the development process.

1. INTRODUCTION

In modern software development, coding and unit testing are performed in interleaved fashion to assure code quality. Current development strategies such as *extreme programming* [2] and *test-driven development* [1] rely heavily on the availability of a test suite to allow a programmer to quickly assess the impact of current 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 the code base and/or edits grow in size, it becomes more difficult to identify the failure-inducing change(s), and tedious manual debugging may be needed.

Our change classification technique relies on the change impact analysis of [20] to find the tests potentially affected by an edit, and to associate with each such test a set of affecting atomic changes with interdependences. These affecting changes are then classified 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). Our goal has been to develop a classification in which *Red* changes are highly likely to be failure-inducing, *Green* changes are highly unlikely to be failure-inducing, and with *Yellow* changes in between. Since it was not clear a priori what classification would work best, we designed four change classifiers and compared their effectiveness.

Our method can also find those changes that are not exercised by the current test suite, which are an indication that additional tests are needed. In addition, by gathering the problematic changes that are associated only with worsening tests, it is possible to determine a subset of the edit that can be committed to the repository safely (i.e., without breaking any tests), even in cases where some tests are failing in a programmer's local workspace. This allows early inclusion of each team member's changes in other team member's codes. Early adoption of edited code avoids inefficient parallel implementations of the same functionality as well as possible conflicts when merging independent changes later in the development process.

Our work was implemented in a tool called *JUnit/CIA*[1], an extension of the popular *JUnit* testing framework (see `www.junit.org`) that is closely integrated with the Eclipse development environment (see `www.eclipse.org`). *JUnit/CIA* relies on our *Chianti* tool [20] 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 four classification methods, and visualizes the classified changes by way of a small extension of *JUnit*'s user-interface.

We evaluated *JUnit/CIA* by using it to find failure-inducing changes in program versions obtained from student projects. On this data, the classifier that maximizes the number of *Red* and *Green* changes combines the best precision and recall (defined in Section 4) with the maximal number of *Green* changes that do not need to be examined. Moreover, all but about 10% of the changes could be committed to the repository without breaking tests. We also experimented with a successive pair of versions of the Daikon system [8]. Here, a different classifier (one that minimized the number of *Red* changes) was very effective at finding small failure-inducing change sets. We conjecture that the difference in preferred classifiers is due to differences in edit size, and in the number of improving and worsening tests. In the student case study, edits were small, and a mixture of improving and failing tests occurred. In the case of Daikon, with a vastly larger code base and number of changes, only two tests were worsening, and none were improving.

The contributions of this paper are the definition of four alternative classification methods, and their implementation in a practical tool. We also report on experiments in which change classificaton successfully finds failure-inducing changes, and subsets of edits that can be committed to a repository without breaking any tests.

[1]This name reflects the fact that the tool extends *JUnit* with features for Change Impact Analysis.

```
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]
}
```

(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);
  }
}
```

(b)

Figure 1: (a) Original and edited version of example program. The original version of the program consists of all program fragments *except* those shown in boxes. The edited version 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.

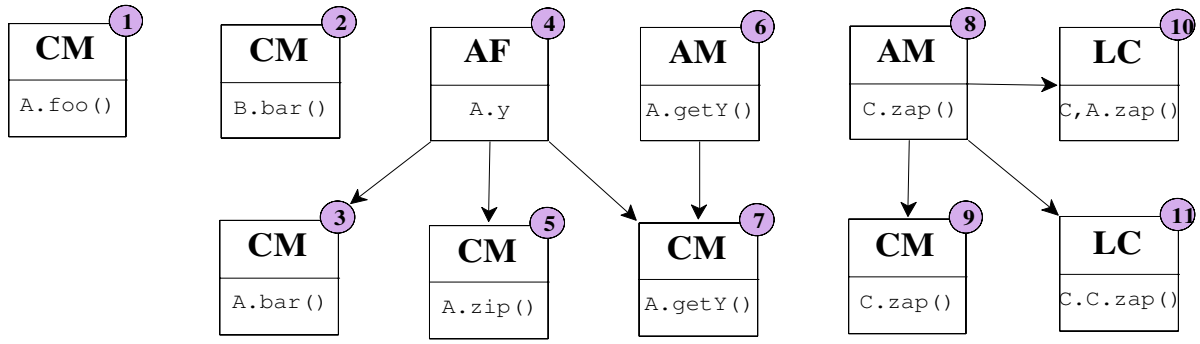

Figure 2: Atomic changes inferred from the two versions of the program.

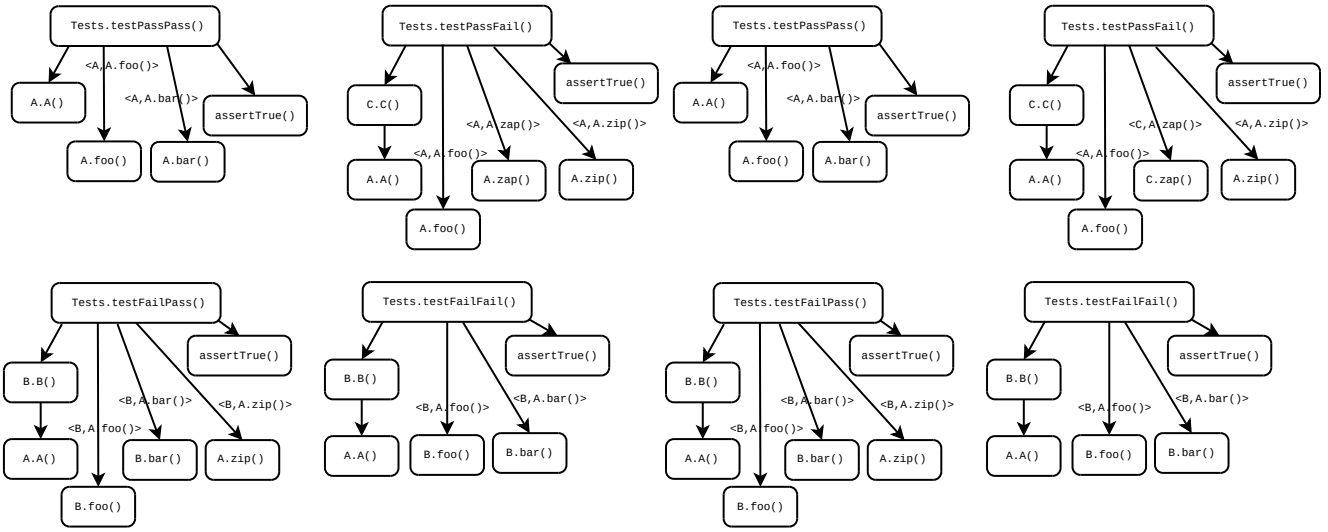

Figure 3: Call graphs for the original version of the program.

Figure 4: Call graphs for the edited version of the program.

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, from which an edited version is obtained by adding all the boxed code fragments. Associated with the program are four *JUnit* tests, `testPassPass`, `testPassFail`, `testFailPass`, and `testFailFail` as shown in Figure 1(b). *JUnit* is a simple, open-source framework that allows users to create and run unit and regression tests. Tests are defined by creating a subclass of the framework class `junit.framework.TestCase` and declaring methods in that class whose name starts with the letters "`test`". These tests can execute arbitrary code, and may perform assertions on computed values by calling framework methods such as `junit.assert.Assert.assertTrue()`. *JUnit* provides a simple user interface for running these tests, and visualizes the test's outcome, which may be success, assertion failure, or exception (crash). When a test failure occurs, the only information provided by *JUnit* is a stack-trace.

We assume that the tests of Figure 1(b) will be used with both the original and edited versions of the program, and the name of each test indicates its outcome in the original and in the modified program. For example, `testPassFail` passes in the original program, but produces an assertion failure in the modified version.

2.1 Atomic Changes

Our change impact analysis relies on the computation of a set of atomic changes that capture all source code modifications at a semantic level that is amenable to analysis. We use a fairly coarse-grained model of atomic changes, with change categories such as as added classes (AC), deleted classes (DC), added methods (AM), deleted methods (DM), changed methods (CM), added fields (AF), deleted fields (DF), and lookup (i.e., dynamic dispatch) changes (LC). A few more atomic change categories that are irrelevant for the example under consideration exist; [20] provides further details.

We also 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). These dependences can be used to construct syntactically valid intermediate versions of the program that contain some, but not all the atomic changes[2].

Figure 2 shows the atomic changes that define the two versions of the example program, numbered 1 through 11 for convenience. 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, both the class and method involved are shown). An arrow from an atomic change A_1 to an atomic change A_2 indicates that A_2 is dependent on A_1. 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 would lead to a syntactically invalid program unless field `A.y` is also added. Therefore, atomic change 5 is dependent on atomic change 4, which is an AF change for field `A.y`.

[2] 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, C_1 and C_2, affect a given test t, then the absence of a syntactic dependence between C_1 and C_2 does not imply the absence of a semantic dependence; that is, program behaviors resulting from applying C_1 alone, C_2 alone, or C_1 and C_2 together, may all be different.

In some cases, a single source code change is decomposed into several atomic changes. For example, the addition of `A.getY()` produced atomic changes 6 and 7, where the former models the addition of an empty method `A.getY()`, and the latter the addition of the return statement to its 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()` would 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 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 used to dispatch to `A.zap()` in the original program. This change in dispatch behavior is captured by atomic change 10. LC changes are also generated in situations where a dispatch relationship is added or removed as a result of a source code change affecting the class hierarchy, such as changing a method from *abstract* to *non-abstract* [20].

2.2 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. Call graphs contain one node for each method, and edges between nodes to reflect calling relationships between methods. Our analysis can work with call graphs that have been constructed using static analysis, or with call graphs that have been obtained by observing the actual execution of the tests. In the experiments reported in this paper, dynamic call graphs are used.

Figure 3 shows the call graphs for the 4 tests of Figure 1(b), before the changes have been applied. In these call graphs, edges corresponding to dynamic dispatch are labeled with a pair $< T, M >$, where T 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 version of the program) either contains a node that corresponds to a CM (changed method) or DM (deleted method) change, or an edge that corresponds to a lookup (LC) change. Using the call graphs in Figure 3, it is easy to see that all four 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 changed method `A.foo()` (change 1), and the call graphs for `testFailPass()` and `testFailFail()` contain nodes corresponding to changed method `B.bar()` (change 2).

2.3 Determining Affecting Changes

In order to compute the changes that affect a given test, we construct a call graph for that test in the *edited* version of the program. Call graphs for the tests 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 (of the edited program), (ii) atomic changes in the lookup change (LC) category that correspond to an edge in the call graph, and (iii) their transitively prerequisite atomic changes.

The *affecting changes* for `testPassFail` can be computed as follows. Observe, that 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 4, 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 requires change 8, and change 5 requires change 4. Therefore, the changes affecting `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`, and that only change 2 affects `testFailFail`.

2.4 Change Classification

Thus far, we have seen that there are 11 atomic changes, and that the behavior of each of the four tests is affected by one or more of these changes. The goal of change classification is to answer the following question: *Which of those 11 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. To a first approximation, this classification works as follows:

- A change that affects only *improving tests*, (i.e., tests such as `testFailPass` that fail in the original program, but that succeed in the modified version) is classified as *Green*.
- A change that affects only *worsening tests*, (i.e., tests such as `testPassFail` that succeed in the original program, but that fail in the modified version) is classified as *Red*.
- A change that affects both improving tests and worsening tests is classified as *Yellow*.

Intuitively, *Red* changes are most likely to be the reason for a test failure, followed by *Yellow* changes, and then *Green* changes.

Of course, there are also tests such as `testPassPass` and `testFailFail` that have the same outcome in both program versions. There are several reasonable ways to classify changes in the presence of such *same-outcome* tests. For example, one could classify a change C as *Green* either if C affects only tests that pass in both program versions, or if all three of the following conditions hold: (i) C affects at least one improving test, (ii) C does not affect any worsening tests, and (iii) C does not affect tests that fail in both program versions. In this approach, *Green* changes are never associated with failing tests, but only with tests that improve and with tests that succeed in both program versions. Alternatively, one could drop condition (iii). Then, changes that affect some tests that fail in both program versions could still be classified as *Green* (here, the argument is that such changes do not degrade any test's result). We will call the first approach the *strict-green* classification, and the second one the *relaxed-green* classification.

Similarly, one could classify a change C as *Red* if: (i) C affects at least one worsening test, (ii) C does not affect any succeeding tests, and (iii) all tests affected by C fail in the edited version of the program. An alternative is to drop condition (iii) because this would prevent changes from being classified as *Red* as soon as they affect a successful same-outcome test. We will refer to the first approach as the *strict-red* classification method, and to the second approach as the *relaxed-red* classification method.

The last column of Table 1 in Section 3 shows the colors of the 11 atomic changes of Figure 2 according to the *strict-green*/*strict-red* classification. Careful examination of the example program reveals that the changes classified as *Red* (changes 8, 9, and 10) are exactly those responsible for the failure of `testPassFail`.

3. DEFINITIONS

In this section, we briefly review the model of atomic changes [20] and then present the new change classification methods.

$$
\begin{aligned}
AT(\mathcal{T},\mathcal{A}) = \; & \{\, t_i \mid t_i \in \mathcal{T},\ Nodes(P,t_i) \cap (\mathbf{CM} \cup \mathbf{DM})) \neq \emptyset \,\} \cup \\
& \{\, t_i \mid \ t_i \in \mathcal{T},\ n, A.m \in Nodes(P,t_i), \\
& \qquad n \rightarrow_{B,\,X.m} A.m \in Edges(P,t_i), \\
& \qquad \langle B, X.m \rangle \in \mathbf{LC},\ B <^* X \,\}
\end{aligned}
$$

$$
\begin{aligned}
AC(t,\mathcal{A}) = \; & \{\, a' \mid a \in Nodes(P',t) \cap (\mathbf{CM} \cup \mathbf{AM}),\ a' \preceq^* a \,\} \cup \\
& \{\, a' \mid \ a \equiv \langle B, X.m \rangle \in \mathbf{LC},\ B <^* X, \\
& \qquad n \rightarrow_{B,\,X.m} A.m \in Edges(P',t), \\
& \qquad \text{for some } n, A.m \in Nodes(P',t),\ a' \preceq^* a \,\}
\end{aligned}
$$

Figure 5: Definitions of Affected Tests and Affecting Changes. Here, $A <^* C$ denotes that A is a subtype of C and that $A \neq C$.

3.1 Affected Tests and Affecting Changes

The first step in our method is to perform a pairwise traversal of the abstract syntax trees of two versions of a program in order to obtain a set of *atomic changes* $\mathcal{A}$, whose granularity is roughly at method level. Atomic changes have *interdependences* which induce a partial ordering $\prec$ on a set of them, with transitive closure $\preceq^*$. We will write $C_1 \preceq^* C_2$ to denote the fact that C_1 is a prerequisite for C_2. Informally, this means that applying change C_2 without also applying change C_1 to the original program would result in a syntactically invalid program.

For a given set $\mathcal{T}$ of regression tests, the analyses of [20] determines a subset $\mathcal{T}'$ of $\mathcal{T}$ of tests that are potentially *affected* by the changes in $\mathcal{A}$. Then, for a given affected test t_i in $\mathcal{T}'$, we determine a subset $\mathcal{A}'$ of $\mathcal{A}$ that contains all the changes that may have affected the behavior of t_i. The equations of Figure 5 (taken from [20][3]) formally define *affected tests* and their corresponding *affecting changes*. These equations assume that the original program P is edited to yield program P', where both P and P' are syntactically correct and compilable. Associated with P is a set of tests $\mathcal{T} = t_1,...,t_n$. The call graph for test t_i on the original program is described by a subset of P's methods $Nodes(P,t_i)$ and a subset $Edges(P,t_i)$ of calling relationships between P's methods. Likewise, $Nodes(P',t_i)$ and $Edges(P',t_i)$ form the call graph of test t_i on the edited program P'. Here, a calling relationship is represented as $D.n() \rightarrow_{B,X.m()} A.m()$, indicating possible control flow from method $D.n()$ to method $A.m()$ due to a virtual call to method $X.m()$ on an object of type B.

Informally, the definition of $AT(\mathcal{T},\mathcal{A})$ states that a test t is affected if one of the nodes in t's call graph in the original program P corresponds to a CM or DM change, or if one of its edges corresponds to an LC change. The definition of $AC(t,\mathcal{A})$ determines the affecting changes for a given affected test t in a similar way, by correlating nodes and edges in the call graph for t in the edited program P' with CM, AM, and LC changes. Here, the $\preceq^*$ ordering is used to include any prerequisite changes as well. In the definitions of Figure 5, 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.

3.2 Classifying Tests

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

[3] In [20], we used the notation $AffectedTests(\mathcal{T},\mathcal{A})$ and $AffectingChanges(t,\mathcal{A})$.

[4] Our approach can easily be adapted to accommodate other test result models with, for example, a single error state or with a more fine-grained error-state model.

Change	*relaxed-green/ relaxed-red*	*relaxed-green/ strict-red*	*strict-green/ relaxed-red*	*strict-green/ strict-red*
1	*Red*	*Yellow*	*Red*	*Yellow*
2	*Green*	*Green*	*Yellow*	*Yellow*
3	*Green*	*Green*	*Green*	*Green*
4	*Yellow*	*Yellow*	*Yellow*	*Yellow*
5	*Yellow*	*Yellow*	*Yellow*	*Yellow*
6	*Gray*	*Gray*	*Gray*	*Gray*
7	*Gray*	*Gray*	*Gray*	*Gray*
8	*Red*	*Red*	*Red*	*Red*
9	*Red*	*Red*	*Red*	*Red*
10	*Red*	*Red*	*Red*	*Red*
11	*Gray*	*Gray*	*Gray*	*Gray*

Table 1: Classification of the atomic changes of Figure 2 according to the 4 methods of Figure 6.

that states that passing tests are preferred over failing tests, and that 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:*

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

Remark: 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.

For a given test t, we will use the notation $R_{old}(t)$ and $R_{new}(t)$ to represent the result of test t in the original version of the program and in the edited version of the program, respectively, where: $R_{old}(t), R_{new}(t) \in \{\text{PASS}, \text{FAIL}, \text{CRASH}\}$. Definition 3.2 below uses these definitions to classify tests as worsening and improving. Tests that are new or that have been deleted in the edited program have no effect on change classification, as they do not correlate with improved or degraded test results.

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:*

$$\begin{aligned} WT &= \{\, t \in \mathcal{T} \mid R_{old}(t) > R_{new}(t) \,\} \\ IT &= \{\, t \in \mathcal{T} \mid R_{new}(t) > R_{old}(t) \,\} \end{aligned}$$

3.3 Classifying Changes

In the definitions below, we will use the notation $AT(C)$ to represent the tests in $\mathcal{T}$ that are affected by atomic change $C \in \mathcal{A}$. Definition 3.3 defines auxiliary functions *Worsening*, *Improving*, *SomeFailFail*, *SomePassPass*, and *OnlyPassPass*. *Worsening* and *Improving* are the sets of changes that affect at least one worsening test, and at least one improving test, respectively. *SomeFailFail* and *SomePassPass* are the sets of changes that affect at least one test that 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).

$$\begin{aligned} \textit{Worsening} &= \{\, C \mid C \in \mathcal{A},\, WT \cap AT(C) \neq \emptyset \,\} \\ \textit{Improving} &= \{\, C \mid C \in \mathcal{A},\, IT \cap AT(C) \neq \emptyset \,\} \\ \textit{SomeFailFail} &= \{\, C \mid \exists t \in AT(C), \\ &\quad R_{old}(t) = R_{new}(t) \in \{\, \text{FAIL}, \text{CRASH} \,\} \,\} \\ \textit{SomePassPass} &= \{\, C \mid \exists t \in AT(C), \\ &\quad R_{old}(t) = R_{new}(t) = \text{PASS} \,\} \\ \textit{OnlyPassPass} &= \{\, C \mid \forall t \in AT(C), \\ &\quad R_{old}(t) = R_{new}(t) = \text{PASS} \,\} \end{aligned}$$

We now can classify changes as *Red*, *Yellow*, or *Green*. Intuitively, our goal is to develop a classification in which *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. As we discussed in Section 2, there are several ways in which one could design such a classification, and it was not clear to us *a priori* which approach would work best in practice. Therefore, our approach will be to define 4 different classifications that each partition the set of changes into *Red*, *Yellow*, and *Green* subsets in slightly different ways. Then, in Section 4 we will present a comparative evaluation of these different classification mechanisms on a set of Java applications with associated *JUnit* tests, in order to determine which classification works best.

The four classification methods of Figure 6 are obtained by using either a "relaxed" or a "strict" criterion for determining *Green* changes, together with a "relaxed" or a "strict" criterion for determining *Red* changes. Thus, we obtain four criteria that we will refer to as *relaxed-green*/*relaxed-red*, *relaxed-green*/*strict-red*, *strict-green*/*relaxed-red*, and *strict-green*/*strict-red*.

Intuitively, the *relaxed-green* 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 the edited version 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 (such a test would also have to fail in the original program, because a change can only be *Green* if it does not affect worsening tests). The *strict-green* criterion eliminates such potentially confusing effects by requiring that all *Green* changes must only affect tests that succeed in the edited version of the program, classifying all changes not meeting this stricter requirement as *Yellow*.

The difference between *relaxed-red* and *strict-red* is similar. The *relaxed-red* classifier marks as *Red* any change that affects worsening tests but not improving tests. This is a reasonable classification, but it may produce the counterintuitive effect that a change that affects a test succeeding in both versions of the program may still be *Red*. The *strict-red* criterion further restricts *Red* changes to only affect tests that fail or crash. Any changes that do not meet this more stringent requirement to be *Red* are classified as *Yellow*.

Some changes do not affect any tests. We classify a change C as *Gray*, if it has no affected tests (i.e. $AT(C) = \emptyset$). 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 1 shows how the changes of the example of Figure 2 are classified according to the 4 classification criteria of Figure 6.

Note that there is an asymmetry in the four change classification methods. A change that affects only tests that pass in both versions is always classified as *Green*, whereas a change for which all affected tests fail in both versions always is 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 can never be failure-inducing and is therefore classified as *Green*. In contrast, if a change C 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 C. Therefore, *Yellow* seems a more appropriate choice than *Red*.

3.4 Committable Changes

In current development practice, it is customary to only release changes to a version control repository when all tests succeed. As a result, commit intervals between versions are often long, with significant differences between successive versions, which complicates the task of integrating changes made by different develop-

$$C \in Green \Leftrightarrow C \in OnlyPassPass \vee (C \in Improving \wedge C \notin Worsening)$$
$$C \in Red \Leftrightarrow (C \notin Improving \wedge C \in Worsening)$$
$$C \in Yellow \Leftrightarrow C \notin Red, C \notin Green, AT(C) \neq \emptyset$$

Classification 1: (*relaxed-green/relaxed-red***)**

$$C \in Green \Leftrightarrow C \in OnlyPassPass \vee (C \in Improving \wedge C \notin Worsening)$$
$$C \in Red \Leftrightarrow (C \notin Improving \wedge C \in Worsening \wedge C \notin SomePassPass)$$
$$C \in Yellow \Leftrightarrow C \notin Red, C \notin Green, AT(C) \neq \emptyset$$

Classification 2: (*relaxed-green/strict-red***)**

$$C \in Green \Leftrightarrow C \in OnlyPassPass \vee (C \in Improving \wedge C \notin Worsening \wedge C \notin SomeFailFail)$$
$$C \in Red \Leftrightarrow (C \notin Improving \wedge C \in Worsening)$$
$$C \in Yellow \Leftrightarrow C \notin Red, C \notin Green, AT(C) \neq \emptyset$$

Classification 3: (*strict-green/relaxed-red***)**

$$C \in Green \Leftrightarrow C \in OnlyPassPass \vee (C \in Improving \wedge C \notin Worsening \wedge C \notin SomeFailFail)$$
$$C \in Red \Leftrightarrow (C \notin Improving \wedge C \in Worsening \wedge C \notin SomePassPass)$$
$$C \in Yellow \Leftrightarrow C \notin Red, C \notin Green, AT(C) \neq \emptyset$$

Classification 4: (*strict-green/strict-red***)**

Figure 6: Definitions of four methods for classifying atomic changes into *Red*, *Yellow*, and *Green* changes.

ers. We will now discuss how the *Worsening* changes set of Definition 3.3 can be used to partition the changes into *committable* and *non-committable changes*, so that the former can be released to a repository safely. Definition 3.4 below formalizes a commit policy that can be characterized informally as: *"Do not commit any changes that break tests."*. We define as committable any change that is uncovered or that only contributes to passing tests, and whose (transitive) prerequisite changes are also committable.

DEFINITION 3.4 (STRICT COMMITTABLE CHANGES).

$$\mathcal{A}_{StrictCommittable} = \{\ C \,|\, t \in AT(C) \Rightarrow R_{new}(t) = \text{PASS}, \quad \forall C' : C' \preceq^* C : C' \in \mathcal{A}_{StrictCommittable}\ \}$$

In practice, the above policy may be overly restrictive because it excludes changes that affect tests that fail in both versions of the program. Definition 3.5 below presents an alternative, more flexible commit policy that could be characterized informally as: *"Do not commit any changes that make things worse."*

DEFINITION 3.5 (COMMITTABLE CHANGES).

$$\mathcal{A}_{Committable} = \{\ C \,|\, C \notin Worsening, \quad \forall C' : C' \preceq^* C : C' \in \mathcal{A}_{Committable}\ \}$$

Note that Definitions 3.4 and 3.5 both state that changes which are *not covered* by any test are committable, unless they have prerequisite changes that are not committable. This allows for situations in which developers are not writing their own tests. Other commit policies can be modeled similarly. For example, a commit policy that demands that all changes released to the repository should be successfully tested could be modeled by only accepting changes C for which $\exists t \in AT(C) \wedge \forall t \in AT(C) : R_{new}(t) = \text{PASS}$.

Consider applying the criterion of Definition 3.4 to the atomic changes 1–11 of Figure 2. Changes 1,2,3,4,5,6,7,11 are either associated with tests that pass in the edited version, or they are not covered. Of these changes, 1,2,4,5 are also associated with with failing tests (i.e., `testFailFail` or `testPassFail`) so they are eliminated. Moreover, the non-committable change 4 is a prerequisite for changes 3 and 7, which implies that change 3 or 7 are not committable either; change 11 has to be excluded for similar reasons. Thus, the only strict committable change in in our example is change 6.

For the criterion of Definition 3.5, our set of candidate changes includes changes 2,3,6,7,11 (i.e., those changes that do not affect `testPassFail`). From this set, 3 and 7 are eliminated because they have non-committable change 4 as a prerequisite (which is associated with the worsening test `testPassFail`), and 11 is eliminated for similar reasons. Consequently, changes 2 and 6 are found to be committable by the criterion of Definition 3.5. In this case, the fact that change 2 affects a test that fails in both versions is no longer a reason to consider it non-committable.

If a set of changes has been identified as non-committable, it may be desirable to *automatically rollback all non-committable changes* to create an intermediate committable version. Working code can then be kept, and changes breaking necessary functionality can be undone, regardless of the (temporal) order in which these changes were originally applied. To further explore such a process, we have developed *Crisp* [4], a tool based on *Chianti* that automatically constructs syntactically valid intermediate program versions that include a user-specified set of atomic changes.

4. IMPLEMENTATION AND EVALUATION

We implemented our change classification method in a tool called *JUnit/CIA*, which is implemented as an Eclipse plugin, and builds on the analysis component of the *Chianti* tool that we developed previously [20]. *JUnit/CIA* uses the version of the program that is currently in the workspace as the "edited version", and retrieves an original version from the local history[5] that corresponds to the last time that the test suite was executed. Dynamic call graphs for the tests are obtained by monitoring their execution using the JVMPI profiling interface.

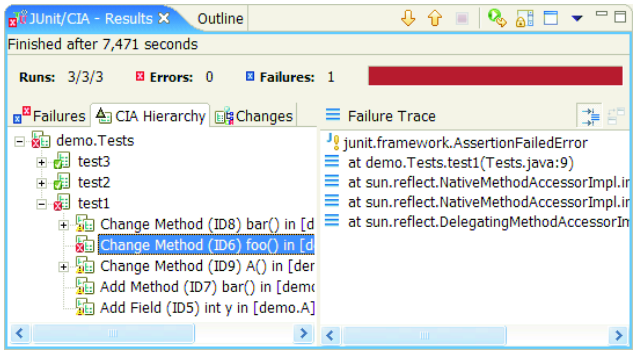

Figure 7: *JUnit/CIA* results view

The user-interface of *JUnit/CIA* extends that of *JUnit* as follows: (i) in the *JUnit* "test hierarchy" view, affecting changes are

[5] The "local history" is a local RCS repository maintained by Eclipse that records all textual changes.

shown in a tree-view underneath each test, where expanding the tree reveals prerequisite changes (see Figure 7), 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 brings up a standard Eclipse compare view of the associated original and edited code.

In order to improve performance, we implemented a filtering mechanism that allows users to avoid the tracing of methods in the standard libraries. Although such methods will never contain any changes, they may execute virtual method calls that dispatch to methods in user code (i.e., callbacks), and such dispatch operations may exhibit changed behavior when overridden library methods are added, deleted, or changed. We conservatively approximate when such behavioral differences may occur by correlating such method changes with the classes instantiated by each test, using an approach similar to that of [29].

During our experiments with student code, 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 these filtering and time-out options.

4.1 Case Study 1: Student Projects

In the first case study, 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 by the implementation in order for students to pass the course. In addition, we defined a number of *secret tests* that performed additional tests on the implementation. The students knew of the existance of a suite of secret tests, but had no knowledge of the details of these tests. Course management was provided using the web-based *Praktomat* system [32]. 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, and we used these versions as the basis for this case study.

Although the students have to agree that their code can be used in research projects, they had no specific knowledge that their data would be used to evaluate *JUnit/CIA*. 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`[6]. Our postprocessing consisted of writing equivalent *JUnit* tests with assertions based on the mandatory interfaces, and adding finer-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.

We analyzed a total of 1175 version pairs written by 40 students[7]. On average, each of the final, graded solutions consisted of 950 LOC of commented Java source code. The total code base analyzed in the experiment was 1240 KLOC. Of the 1175 version pairs, only 669 contained semantic changes[8]. Specifically, these 669 pairs contained a total of 7522 atomic changes, of which 5494 (i.e., 73%) were *Gray*[9]. The distribution of the remaining covered 2028 changes strongly depends on the classifier used, and is shown in Table 2. For example, the *relaxed-green*/*relaxed-red* classifier finds 1053 *Green* changes, 765 *Yellow* changes, and 210 *Red* changes. The last column in the table, labeled "**%***Green*", is simply the percentage of covered changes that are classified as *Green*. In general, it is desirable to have this percentage as high as possible, so that there are fewer *Red* and *Yellow* changes for programmers to focus on.

Classifier	*Green*	*Yellow*	*Red*	*%Green*
relaxed-green/*relaxed-red*	1053	765	210	52%
strict-green/*relaxed-red*	554	1264	210	27%
relaxed-green/*strict-red*	1053	846	129	52%
strict-green/*strict-red*	554	1345	129	27%

Table 2: Classification of covered changes.

We will now evaluate the effectiveness of the four classifiers at correctly identifying failure-inducing changes by classifying them as *Red*. Of the 669 version pairs with semantic changes, only 105 involved worsening tests. Note that only version pairs with worsening tests are of interest here, because in other cases there are no (additional) failures to investigate. To this end, we manually identified the failure-inducing change sets (FICS) in 100 of the 105 version pairs, by selectively undoing subsets of changes and observing whether or not the failure still occurred (in the remaining 5 version pairs, we were unable to determine the failure-inducing changes due to the large size of edit). Associated with these 100 version pairs were a total of 439 worsening tests. We found that a total of 112 distinct failure-inducing change sets were responsible for these worsening tests (i.e., on average, a single failure-inducing change set was associated with 3.9 worsening tests).

Table 3 states a number of statistics that evaluate the effectiveness of the four classification methods of Figure 6 at identifying failure-inducing change sets. The columns labeled **R**, **Y**, and **G**, respectively, show how many of the 112 failure-inducing change sets are classified as *Red*, *Yellow*, and *Green*, respectively. Here, a failure-inducing change set is classified as *Red Yellow*, or *Green*, if at least 50% of the changes contained in that set were given that color, and if no other changes were given a "stronger" color (e.g., the number of FICSs reported as *Yellow* excludes situations where *Red* changes were reported outside the FICS). For example, the *strict-green*/*strict-red* method classifies 45 failure-inducing change sets as *Red*, 61 as *Yellow*, and 0 as *Green*, respectively. The next column, labeled "**Miss**", shows the number of times where a failure-inducing change set was identified as *Yellow*, although *Red* changes were reported outside the FICS. In other words, this column indicates the number of times where the classification actively points the user at the wrong change(s). The *strict-green*/*strict-red* classifier produced 6 such misclassifications.

The columns labeled **Recall** and **Precision** in Table 3 are key

[6] DejaGNU is an open-source black box regression-testing framework, see http://www.gnu.org/software/dejagnu/.

[7] This excludes 3 solutions with non-deterministic test behavior.

[8] Our analysis considers two versions the same if they only differ in terms of layout or comments. The relatively high number of versions without changes is due to coding style requirements for the course. Unfortunately, our students tend to first write working code and add comments, improve layout, etc. afterwards, which results in many different versions without functional changes.

[9] The high percentage of uncovered changes is due to the relatively coarse-grained test suite, and the fact that several students first implemented the functionality and then adapted their code to match the obligatory interfaces, so that our unit tests failed to actually access the implemented functionality for intermediate versions.

Classifier	FICS classified as				Recall	Precision
	R	Y	G	Miss		
relaxed-green/relaxed-red	81	30	0	1	72%	99%
strict-green/relaxed-red	81	30	0	1	72%	99%
relaxed-green/strict-red	45	61	0	6	40%	88%
strict-green/strict-red	45	61	0	6	40%	88%

Table 3: Comparison of the effectiveness of the Classifiers.

concepts from information retrieval [9] that are defined as follows:

DEFINITION 4.1 (RECALL AND PRECISION).

$$Recall = \frac{\#correct\ items\ retrieved}{\#items\ to\ find}$$

$$Precision = \frac{\#correct\ items\ retrieved}{\#items\ retrieved}$$

Informally, *Recall* states how many of the desired results were found, while *Precision* estimates how many of the retrieved items were actually desired. For the purposes of this case study, *# items retrieved* is the sum of the number of failure-inducing change sets correctly identified as *Red* and the number of misclassified failure-inducing change sets, *# correct items retrieved* is the number of failure-inducing change sets correctly identified as *Red*, and *# items to find* is the total number of failure-inducing change sets. For example, for either of the *relaxed-red* classifications, we have a recall of 81/112 = 72%, and a precision of 81/82 = 99%, and for either of the *strict-red* classifications, we have a recall of 45/112 = 40% with a precision of 45/51 = 88%. From the results in Tables 2 and 3, it is evident that the *relaxed-green*/*relaxed-red* classifier should be preferred, because it combines the superior recall and precision of the *relaxed-red* classifier with the high percentage of *Green* changes of the *relaxed-green* classifier. The surprising larger number of misclassifications by the *strict-red* classifier is due to the fact that it colors some failure-inducing changes *Yellow* while some *Red* changes that are unrelated to the failure remain. Note that the *relaxed-green* classifier optimistically assumes that a change that affects a test that fails in both versions of the program does not contribute to this failure. This appears to be a reasonable assumption, as we never encountered such situations during the experiments.

We analyzed several specific version pairs of the student projects in more detail. Most of these contained only a handful of affecting changes per test due to the small commit intervals. In these cases, the *relaxed-red* classifier typically identified the failure-inducing changes as *Red*, and the remaining few affecting changes for that test are usually *Red* or *Yellow*. One of the few slightly more interesting cases is version pair *Stud 3/22/23*, each of whose 8 worsening tests is affected by the same 10 atomic changes. Here, we manually determined that a single CM change to method `Queue.insert()` caused the failure. The *relaxed-green*/*relaxed-red* classifier correctly identified this change as *Red*, and 7 of the remaining 9 affecting changes were classified as *Yellow*.

We also counted the committable changes in the student code, according to Definition 3.5. Only 762 of all 7522 changes (i.e., about 10%) failed to be committable. Although a considerable number of these changes may not have been intended to be released to a repository, this nevertheless demonstrates the value of change classification in collaborative development scenarios.

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 could not find any worsening unit tests. 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 new 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 66 tests were defined, of which 47 were affected by the edit. The two versions differed significantly, as a total of 6081 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 *relaxed-red* classifier failed to focus on these changes because it classified all 35 affecting changes as *Red*, as there are no improving tests for these two versions. Both *strict-red* 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 *strict-red* classifiers were very successful at correctly focusing the programmer's attention on the appropriate affecting changes.

The second test, `testMinus`, produced a similar result. This test was affected by 34 changes, and we manually identified the failure-inducing changes to be a CM change to method `daikon.diff.Diff.shouldAdd()`. Again, the *relaxed-red* classifier was not useful because it classified all 34 changes as *Red*. The *strict-red* classifiers were again 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.

The two Daikon versions were separated by 6081 changes of which 5705 were classified as *Gray* due to the low coverage of the Daikon unit test suite. Of the remaining 376 changes, 338 were *Green*, 31 *Yellow*, and only 7 *Red* using the *strict-green*/*strict-red* classifier variant. 6025 of the 6081 changes were classified as committable, according to Definition 3.5.

4.3 Assessment

While it seems contradictory that the case studies involving student data and Daikon suggest that different classification methods should be preferred, we think there is a good reason for this. The student projects are characterized by small differences between versions, and a mixture of improving and worsening tests can be observed. We consider this to be very close to the intended usage scenario of our system in an IDE such as Eclipse where unit tests are executed frequently, and it is encouraging to see that failure-inducing change sets can be determined with relatively high precision and recall. In the Daikon study, on the other hand, where the differences between versions and the sets of changes affecting each test tend to be much larger, there are only a few worsening tests, and no improving tests. In this case, it is encouraging to see that the *strict-red* classification allows one to quickly isolate the failure-inducing changes among the changes affecting each test. While these case studies present a number of interesting data points, more empirical studies on large real-world programs are clearly needed.

5. RELATED WORK

Delta Debugging. In the work by Zeller et al. on *delta debugging*, the reason for a program failure is identified as a set of differences between versions [31], inputs [34], thread schedules [5], or program states [33, 6] 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 correlating the outcome of each execution (*pass*, *fail*, or *inconsistent*), the set of failure-inducing changes can be narrowed down using efficient binary-search techniques.

Our work and delta debugging both aim at identifying failure-inducing changes, but differ in several important ways. Delta debugging may construct an intermediate program that is syntactically invalid or produces indeterminate results, and therefore requires an *inconsistent* test outcome. We assume programs to be compilable when tests are executed and our tests have outcomes that are determinate, (PASS, FAIL, CRASH). Delta debugging determines if change is failure-inducing or not by examining the effect of its presence or absence on two program versions. Our use of a more fine-grained (i.e., *Red*, *Yellow*, *Green*, *Gray*) classification of changes stems from our observation of the effect of changes on multiple tests. Delta debugging requires the execution of intermediate programs in order to find failure-inducing changes, which may require, in the worst case, a number of executions proportional to the number of changes. In our approach, which does not require the execution of intermediate programs, the construction of dynamic call graphs involves only a constant overhead factor at run-time. Delta debugging may be able to narrow down the set of failure-inducing changes more effectively, because the execution of intermediate program versions provides additional information about the reason for a program failure. 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, with different strengths and weaknesses, may complement each other. In principle, the use of a richer model of changes with interdependences could improve the efficiency of delta debugging by reducing the number of intermediate programs that need to be constructed. Conversely, our change classification could be made more precise by executing tests on intermediate program versions, and taking their results into account.

Comparing Dynamic Data Obtained From Different Executions. Several debugging approaches rely on comparing dynamic information associated with succeeding and failing runs. Reps et al. [22] proposed comparing path profile data obtained from different program executions in order to expose incorrect Year 2000 date-related computations that might give rise to the execution of different paths. Harrold et al. [11] evaluated 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. [12] present a *discrete* visualization approach in which the colors red, yellow, and green are used to visualize statements executed only by failing tests, by succeeding and failing tests, and only by succeeding tests, respectively. Jones et al. remark that this approach is "not very informative, as most of the program is yellow". Because of the inconclusive nature of these results, Jones et al. also developed a *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. The main difference between our work and their discrete approach is that 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 effective because the number of executed changes is generally far smaller than the number of executed statements. Moreover, the execution of different statements by a failing test may have been caused by a change in a completely different location due to the non-locality of change impact in object-oriented programs, so focusing the programmer's attention at such changes is likely to be more helpful. It would be interesting to experiment with the use of a continuous scale of color and brightness in our work as well; we consider this a possible topic for future work.

Liblit et al. [16, 17] 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. The work of [16] applies in situations with a single bug, whereas that of [17] can identify separate causes in the presence of multiple bugs.

Dallmeier et al. [7] present a technique for localizing errors by comparing sequences of method calls in passing and failing runs of a program. From their experiments they conclude that comparing method call sequences is a better defect indicator than a simple coverage-based metric for method calls, such as the one by Jones et al [12], and that comparing sequences of method calls *on the same object* is an even better predictor.

Change Impact Analysis. We previously presented the conceptual framework [23] of our change impact analysis, and its expansion to the full Java language with empirical validation [21, 20]. The goals of this work is to find *a subset of the changes* that impact a given test, and to *classify* changes based on their impact on test behaviors. Other research on impact analysis has concentrated on finding *program constructs* potentially affected by changes. These analyses are based on static analysis [3, 15, 13, 30], dynamic analysis [14] or, like our analysis, on a combination of the two [18]. Recent work on change impact analysis includes the *PathImpact* algorithm by Law and Rothermel [14], 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. [18], which combines the use of a forward static slice [28] with respect to 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 can be found in [19].

Continuous Testing and Test Factoring. Saff and Ernst [24, 27] propose to use the idle CPU during editing for safe asynchronous *continuous testing*. Their experiments derive from recorded development data a positive correlation between the time span from bug introduction to bug discovery (ignorance time), and the amount of time required to fix bugs (fix time), in order to demonstrate the benefit of continuous testing based on a cognitive model of developer actions. In [26], these results are confirmed in a comparative case study with student developers. The same authors introduce *test factoring* [25], a technique for automatically creating fast, focused unit tests from slow system-wide tests using dynamic analysis, with the objective of reducing the amount of time until test failures occur (wait time). Change classification complements the work on continuous testing and test factoring by directly reducing fix time.

6. CONCLUSIONS

There are four main contributions of this paper. First, using change impact analysis we find affecting changes for each possibly affected test of a edited Java program. We defined four different change classifiers that identify affecting changes as *Red*, *Yellow*, or *Green*, according to the (increasing) likelihood that

they are failure-inducing. Second, we implemented this change classification methodology in *JUnit/CIA*, an Eclipse plugin built on *Chianti* and the *JUnit* testing framework. Third, we have obtained initial promising results from two case studies indicating that our classification can help in focusing programming attention on possible failure-inducing changes. Although we were unable to conclusively determine which classifier is the best—this will require more investigation—the *relaxed-green*/*relaxed-red* classifier had the best recall and precision (student study), while the *relaxed-green*/*strict-red* classifier was better at identifying the failure-inducing change sets (Daikon study).

Fourth, using our change classification, we outlined two distinct policies to determine subsets of changes that can be committed safely to a repository without breaking tests. Our experiments revealed that most changes can be committed safely, even in cases where a programmer's local workspace contains failing tests.

7. REFERENCES

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

[2] BECK, K., AND FOWLER, M. *Planning Extreme Programming*. Addison-Wesley, 2001.

[3] 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.

[4] CHESLEY, O., REN, X., AND RYDER, B. G. Crisp: A debugging tool for Java programs. Tech. Rep. DCS-TR-05-567, Department of Computer Science, Rutgers University, April 2005.

[5] CHOI, J.-D., AND ZELLER, A. Isolating failure-inducing thread schedules. In *Proc. ACM SIGSOFT International Symp. on Softw. Testing and Analysis (ISSTA 2002)* (Rome, Italy, 2002), pp. 210–220.

[6] CLEVE, H., AND ZELLER, A. Locating causes of program failures. In *Proc. 27th International Conf. on Softw. Engineering (ICSE 2005)* (St. Louis, MO, 2005). To appear.

[7] 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). To appear.

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

[9] FISCHER, B. *Deduction-Based Software Component Retrieval*. PhD thesis, Fakultät für Methemeatik und Informatik, University of Passau, 2001.

[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., HARROLD, M. J., AND STASKO, J. Visualization of test information to assist fault localization. In *Proc. International Conf. on Softw. Engineering (ICSE'02)* (Orlando, FL, 2002), pp. 467–477.

[13] 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 International Conf. on Softw. Maintenance* (1994), pp. 202–211.

[14] LAW, J., AND ROTHERMEL, G. Whole program path-based dynamic impact analysis. In *Proc. of the International Conf. on Softw. Engineering* (2003), pp. 308–318.

[15] LEE, M., OFFUTT, A. J., AND ALEXANDER, R. T. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proc. 34th International Conf. on Technology of Object-Oriented Languages and Systems (TOOLS USA'00)* (Santa Barbara, CA, 2000).

[16] 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.

[17] 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). To appear.

[18] 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).

[19] ORSO, A., APIWATTANAPONG, T., LAW, J., ROTHERMEL, G., AND HARROLD, M. J. An empirical comparison of dynamic impact analysis algorithms. In *Proc. of the International Conf. on Softw. Engineering (ICSE'04)* (Edinburgh, Scotland, 2004), pp. 491–500.

[20] 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.

[21] REN, X., SHAH, F., TIP, F., RYDER, B. G., CHESLEY, O., AND DOLBY, J. Chianti: A prototype change impact analysis tool for Java. Tech. Rep. DCS-TR-533, Rutgers University Department of Computer Science, September 2003.

[22] 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.

[23] 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).

[24] SAFF, D., AND ERNST, M. D. Reducing wasted development time via continuous testing. In *Fourteenth International Symp. on Softw. Reliability Engineering* (Denver, CO, November 17–20, 2003), pp. 281–292.

[25] 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.

[26] SAFF, D., AND ERNST, M. D. An experimental evaluation of continuous testing during development. In *ISSTA 2004, Proc. of the 2004 International Symp. on Softw. Testing and Analysis* (Boston, MA, USA, July 12–14, 2004), pp. 76–85.

[27] SAFF, D., AND ERNST, M. D. Continuous testing in eclipse. In *Proc. of the 26th International Conf. on Softw. Engineering (ICSE'05)* (St. Louis, MO, USA, May 2005).

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Trans. on Softw. Eng. 28*, 2 (2002), 183–200.