RC23729 (W0509-100) September 29, 2005
Computer Science

IBM Research Report

Finding Failure-Inducing Changes Using Change Classification

Maximilian Stoerzer
Lehrstuhl Softwaresysteme
University of Passau
Innstraße 32, 94032 Passau
Germany

Barbara G. Ryder, Xiaoxia Ren
Department of Computer Science
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854

Frank Tip
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Finding Failure-Inducing Changes 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 are not always easy to find. Change classification focuses programmer attention on those changes most likely to be failure-inducing. We define several classifiers that automatically label changes as *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. Initial results indicate that change classification succeeds in focusing programmer attention on failure-inducing changes, thus improving on manual debugging techniques. 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.

1. INTRODUCTION

In modern software development, coding and unit 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 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 [17] 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). Our goal has been to develop a classifier 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. Since it was not clear *a priori* which classifier would work best, we designed five change classifiers and compared their effectiveness.

In addition, by gathering problematic changes that are associated only with tests whose outcome has degraded, it is possible to determine a subset of the edit that can be committed to a 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 adoption of each team member's changes by other team members, which helps prevent inefficient parallel implementations of the same functionality and reduces the number of possible conflicts when merging changes later in the development process.

Our prototype *JUnit/CIA*[1] is an extension of the Eclipse component that integrates the popular *JUnit* testing framework with the Eclipse IDE (see `www.junit.org` and `www.eclipse.org`). *JUnit/CIA* relies on *Chianti* [17] 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 the user-interface in *JUnit*.

We conducted two case studies with *JUnit/CIA* to find failure-inducing changes in student programs and in *Daikon* [6], respectively. In each study, the actual causes of failure (i.e., failure-inducing changes) were determined by human examination of the code. The student programs study determined a *best* classifier (*relaxed-red/relaxed-green*) from among five candidates that was superior in focusing programmer attention on the failure-inducing changes. Surprisingly, the *Daikon* study found a different classifier (*strict-red/relaxed-green*) to be most effective. This difference is due to properties of the data used in the two studies (see Section 4).

The main contributions of this paper are the demonstration that change classification is (i) an approach to focus programmer attention effectively on likely sources of failure and (ii) an enabler of the early release of changes to a repository, aiding early adoption of new code by team members. In addition, the empirical case studies provide quantitative measurement of the effectiveness of several classifiers on different kinds of programs. For example, in the student programs study, there were 444 tests with two or more affecting changes, whose outcomes were degraded (by the edit); the *relaxed-red/relaxed-green* classifier focused programmer attention on the failure-inducing changes correctly in 47.5% of these tests. Of the 6624 changes in this entire case study, 3553 (53.6%) were found to be safe candidates for release to the repository.

2. EXAMPLE OF OUR APPROACH

Figure 1(a) shows two versions of a small example program.

[1]This name reflects the fact that the tool extends the functionality of *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]
}
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.

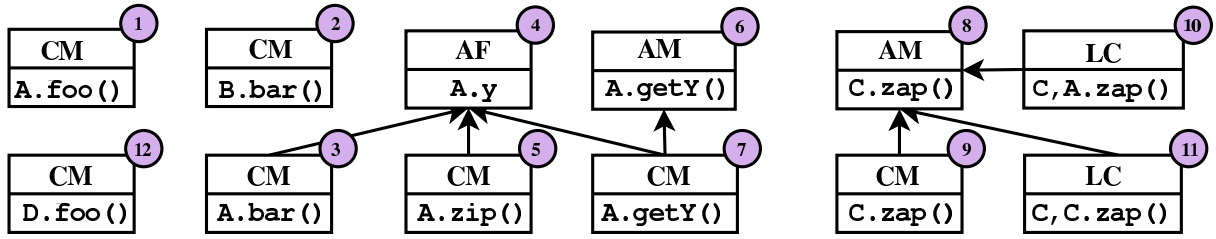

Figure 2: Atomic changes inferred from the two versions of the program.

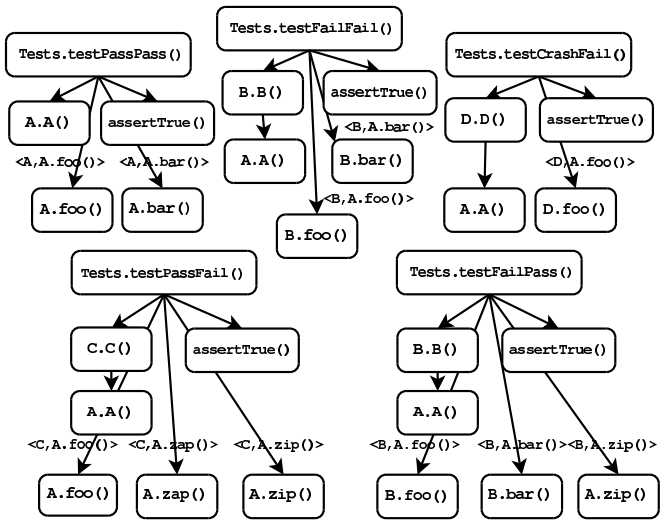

Figure 3: Call graphs for the original version of the program.

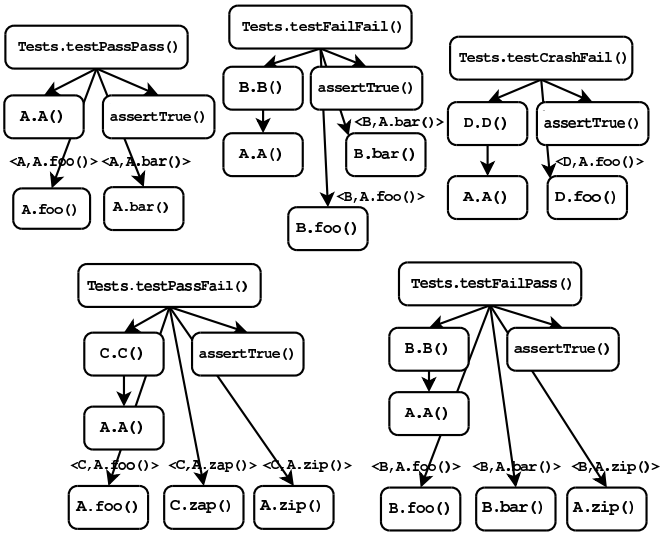

Figure 4: Call graphs for the edited version of the program.

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.

Atomic Changes. Our change impact analysis 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 methods (**CM**), added fields (**AF**), deleted fields (**DF**), and lookup changes (**LC**) (i.e., dynamic dispatch).[2]

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, $A_2 \preceq A_1$). Formally, syntactic dependences define a partial ordering $\preceq$ on a set of changes, with transitive closure $\preceq^*$. These dependences can be used to construct syntactically valid intermediate versions of the program that contain some, but not all the atomic changes.

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. 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.

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.[3] Our analysis can work with call graphs that have been constructed either using static analysis, or by observing the actual execution of the tests.

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 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 requires 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, this 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 edited version) is classified as *Green*. For example, change 12 (**CM** for `D.foo()`) only affects `testCrashFail` and thus is *Green*.[4]

[2] See [17] for additional change categories.

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

[4] 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, and then *Green* changes. 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 (*strict-red/relaxed-green*, *strict-red/strict-green*), only changes 8, 9 and 10 are colored *Red*. The observant reader may verify that these are exactly the failure-inducing changes for this example.

3. DEFINITIONS

In this section, we present several change classifiers. We implicitly make the usual assumptions [7] that program execution is deterministic and that the library code and execution environment (e.g., JVM) remain unchanged. We will also assume that no dependences between tests exist.[5]

3.1 Classifying Tests and Changes

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 exception is caught by the *JUnit* runtime). Definition 3.1 below formalizes this test result model[6] 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 the notation $R_{orig}(T)$ 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. 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:*

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

[5] While it is possible to create such dependences in *JUnit*, this is often an indication of bad programming style.

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

In the definitions below, we will use the notation $AT(A)$ to represent the tests in $\mathcal{T}$ affected by atomic change $A \in \mathcal{A}$ 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).

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

We now can classify changes as *Red*, *Yellow*, or *Green*. Intuitively, our goal is to develop a classifier for 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. 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. Therefore, 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 on a set of Java applications with associated *JUnit* tests.

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.

The remaining four classifiers depend on the *development of test results* for the two versions. Figure 5 shows how these classifiers 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 classifiers that we will refer to as *relaxed-red/relaxed-green*, *strict-red/relaxed-green*, *relaxed-red/strict-green*, and *strict-red/strict-green*.

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. In the example in Figure 1, change 2 affects both `testFailPass`, an improving test, and `testFailFail`; it will be colored *Green* by the *relaxed-green* criterion. The *strict-green* criterion eliminates such potentially confusing effects by requiring that all *Green* changes must only affect tests that succeed in the edited program, (i.e., change 2 will be colored *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 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 *strict-red* criterion further restricts *Red*

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

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

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

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

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

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

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

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

Figure 5: Definitions of four methods for classifying atomic changes into *Red*, *Yellow*, and *Green* changes.

changes to affect only tests that fail or crash. Any changes that do not meet this more stringent requirement to be *Red* are classified as *Yellow*.

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 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 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 has no affected 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 1 shows how the changes of the example of Figure 2 are classified according to our five classifiers.

Chg.	*simple*	*relaxed-red* *relaxed-green*	*strict-red* *relaxed-green*	*relaxed-red* *strict-green*	*strict-red* *strict-green*
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*	*Yellow*	*Green*	*Yellow*

Table 1: Classification of the atomic changes of Figure 2 according to the *simple* classifier and the 4 classifiers in Figure 5.

3.2 Determining Committable Changes

In current development practice, it is customary to release changes to a version control repository only when all tests succeed. As a result, the intervals between *commits* of changes to a repository are often long, and significant differences may exist between successive versions. In the presence of multiple developers, the existence of significant changes between successive versions may complicate the task of integrating these changes. Determining *committable changes* that can be exposed safely to others by early release of changes to a repository enables developers to reduce the amount of time spent subsequently on change integration.

Before we can determine subsets of changes that can be committed safely in the presence of failing tests, we need to resolve when a set of changes should be considered committable. One possibility is having a commit policy:

Changes that break tests should not be committed.

However, this policy is often unnecessarily restrictive. Consider a situation where a test T fails in both the original and the edited version of the program, and where change A affects T. Then, the failure of T in the edited program may be caused by A, or it may be due to the same reason that caused T's failure in the original version.[7]

It is often sufficient to ensure that no *additional* test failures occur due to committed changes, which corresponds to the following less restrictive commit policy:

Changes that make test outcomes worse should not be committed.

Given this policy, it is clear that any change A that affects a worsening test cannot be committed. In addition, if A is a prerequisite for another change A', then committing A' without also committing A leads to a syntactically invalid program. Consequently, any change A' directly or indirectly depending on non-committable change A must be excluded from the set of committable changes as well. Moreover, as we already remarked in Section 2, *semantic dependences* may also exist between changes. Figure 6 shows an example that illustrates why semantic dependences must be taken into account when determining committable changes.

The example consists of a class `SD` containing methods `SD.zip()` and `SD.zap()`, and a class `Test` that defines tests `test1` and `test2`. The original program consists of all code fragments except those shown in boxes, which are added to obtain the edited program. In the original version, `SD.zip()` and `SD.zap()` are both defined as the identity function, hence both tests succeed. In the edited version, `test2` fails because the value computed for `x` is `2`, which is different from the expected value of `1`. We compute two **CM** changes for this program, one for each of the methods `SD.zip()` and `SD.zap()`, with neither change syntactically dependent on the other. We find that `test1` is affected by both **CM** changes, and that `test2` is only affected by **CM** for `SD.zip()`. However, committing only **CM** for `SD.zap()` would result in a version where `test1` fails, illustrating that semantic dependences must be taken into account when computing committable changes.

Definition 3.4 states that a change A is committable if: (i) A does not affect any worsening tests, (ii) all of A's prerequisite changes

[7] Determining why T fails in the edited program is beyond the scope of the analysis in this paper.

```
class SD {
  static int zip(int x) { x = 2 * x; return x; }
  static int zap(int x) { x = x / 2; return x; }
}
class Test {
  public void test1() {
    int x = 1; x = SD.zip(x); x = SD.zap(x);
    assertEquals(1, x); }
  public void test2() {
    int x = 1; x = SD.zip(x);
    assertEquals(1, x); }
}
```

Figure 6: Example program that illustrates a semantic dependence between two changes. The original program contains all code fragments except those shown in boxes; the edited program is obtained by adding the boxed code fragments.

are committable, and (iii) for any test T affected by A, all of T's affecting changes must be committable. Condition (iii) encodes the conservative assumption that semantic dependences may exist between any pair of changes that affect a given test. This condition means that for each test, either all or none of its affecting changes will be committed. While the suggested definition is safe, it might be unnecessarily imprecise. We consider finding a more precise approximation for semantic dependences to be a topic for future work.

DEFINITION 3.4 (COMMITTABLE CHANGES).

$$\mathcal{A}_{Committable} = \{\ A \,|\, A \notin Worsening, \forall A' : A' \preceq^* A : A' \in \mathcal{A}_{Committable}, \\ \forall T \in AT(A) : A'' \in AC(T) \implies A'' \in \mathcal{A}_{Committable}\ \}$$

Returning to the example program of Figure 1, we can compute the committable changes as follows. Our set of candidate changes includes changes 2, 3, 6, 7, 11 and 12 (i.e., those changes that do not affect `testPassFail`). From this set, 3, 7 and 11 are eliminated because of non-committable prerequisite changes. Consequently, only changes 2, 6 and 12 are candidates for committable changes. Change 2 affects two tests, `testFailFail` and `testFailPass`. `testFailPass` is also affected by non-committable changes 4 and 5. Since we conservatively assume semantic dependences among these changes, change 2 is not committable. Change 6 does not affect any test and change 12 affects the improving test `testCrashFail`, for which it is the only affecting change. Therefore, both changes 6 and 12 are committable.

To further explore the handling of committable changes, we have developed *Crisp* [2], a tool that automatically constructs syntactically valid intermediate program versions that include a user-specified set of atomic changes.

4. IMPLEMENTATION AND EVALUATION

To evaluate our change classifiers we created the tool *JUnit/CIA*, which is implemented as an Eclipse plug-in, and builds on the analysis component of the *Chianti* tool that we developed previously [17]. *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[8] 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.

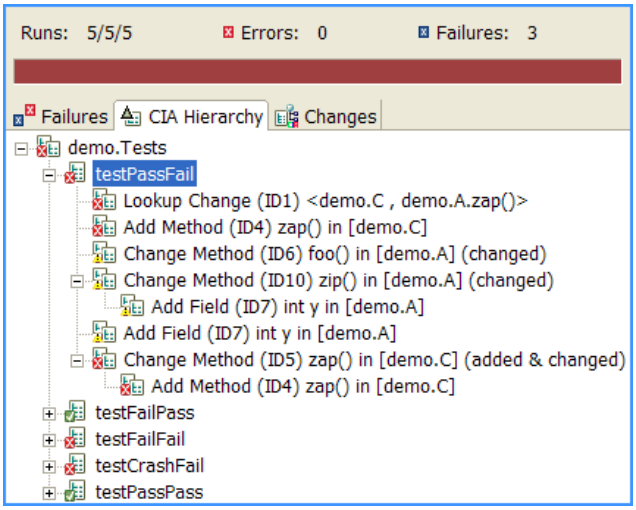

Figure 7: *JUnit/CIA* hierarchy view

The user-interface of *JUnit/CIA* extends that of the *JUnit* Eclipse component as follows: (i) in the *JUnit* test hierarchy view, affecting changes are 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 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 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 [25].

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 in order for students to pass the course. We also defined a *secret tests suite* that performed additional tests on student programs. The students knew about this secret test suite, but had no detailed knowledge about the tests themselves. 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 manage-

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

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

Table 2: Selection of version pairs from the student data.

ment was provided using the web-based *Praktomat* system [28]. 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`.[9] 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.

On average, each of the final, graded solutions consisted of 950 LOC of commented Java source code. We analyzed a total of 1175 version pairs written by 40 students. The total code base analyzed in the experiment was 1240 KLOC. Of these 1175 version pairs, 556 contained semantic changes,[10] and 110 of these 556 version pairs had worsening tests associated with them. For 98 of these 110 version pairs, we could manually identify a failure-inducing change set.[11] Since we are interested in techniques for automatically determining failure-inducing change sets, 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 2.

Per-Version-Pair Evaluation. The 61 version pairs contained a total of 401 atomic changes. Table 3 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*, and *Green*, respectively. For example, the *relaxed-red/relaxed-green* classifier finds 138 *Red*, 126 *Yellow* and 137 *Green* changes.

To determine classifier quality, we manually identified the failure-inducing changes, by selectively undoing subsets of changes and observing whether or not the failure still occurred. For each of the 5 classifiers, we then calculated the *recall* and *precision*. These are core metrics from information retrieval theory,

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

[10] 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.

[11] 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.

Classifier	*#Red*	*#Yellow*	*#Green*
relaxed-red/relaxed-green	138	126	137
strict-red/relaxed-green	77	187	137
relaxed-red/strict-green	138	200	63
strict-red/strict-green	77	261	63
simple	119	238	44

Table 3: Coloring of changes according to the 5 classifiers (cumulative statistics over 61 version pairs).

Classifier	*recall Green*	*prec. Green*	*recall Red*	*prec. Red*	*false Pos.*	*false Neg.*
relaxed-red/ relaxed-green	19.3	100.0	58.5	76.5	23.5	41.5
strict-red/ relaxed-green	19.3	100.0	28.1	87.7	12.3	71.9
relaxed-red/ strict-green	14.5	100.0	58.5	76.5	23.5	41.5
strict-red/ strict-green	14.5	100.0	28.1	87.7	12.3	71.9
simple	12.5	100.0	28.1	76.0	24.0	71.9

Table 4: Recall, precision, false positives, and false negatives for each classifier (average percentages over 61 version pairs).

stating the percentage of desired results retrieved, and the percentage of correctly retrieved items among all retrieved items, respectively. For *Red* changes, *recall* is the percentage of failure-inducing changes colored *Red*, and *precision* is the percentage of actually failure-inducing *Red* changes among all *Red* changes. For *Green* changes, *recall* is the percentage of non-failure-inducing changes colored *Green*, whereas *precision* is the percentage of non-failure-inducing *Green* changes among all *Green* changes.

For *Red* changes,[12] we also computed the average rate of *false positives* (i.e., changes that are classified as *Red* but that are not failure-inducing) and *false negatives* (i.e., failure-inducing changes that are not classified as *Red*). Table 4 shows, on average over the 61 version pairs, the recall and precision of the *Green* changes, the recall and precision of the *Red* changes, and the average percentage of false positives and false negatives, respectively. For an overview of the non-aggregated data, the reader is referred to appendix A (and following).

When analyzing the results of Table 4, it is easy to see that the *simple* classifier can be dismissed, because it has both the lowest recall and the lowest precision. For the 4 other classifiers, the choice between *strict* vs. *relaxed* can be made independently for the *Green* changes and the *Red* changes, so we will consider these decisions separately.

It is not obvious *a priori* whether the *strict-red* or *relaxed-red* classifiers should be preferred. Note that finding the optimal choice is a two-dimensional optimization problem: Ideally, we would like to classify all failure-inducing changes as *Red*, while not coloring any non-failure-inducing changes *Red*. As Table 4 shows, the *strict-red* classifiers yield a better precision (87.7% vs. 76.5%), but the *relaxed-red* classifiers yield a better recall (58.5% vs. 28.1%). In this case, the *relaxed-red* classifiers seem preferable given that they offer significantly ($\sim$30%) higher recall at the cost of a moderate ($\sim$10%) loss in precision.

The choice between the *strict-green* and *relaxed-green* classifiers is easier to make. Examining the results of Table 4, we can ob-

[12] False positives and false negatives only pertain to *Red* changes, and not *Green* changes as we focus on the ability of our classifiers to find failure-inducing changes. Note that the percentage of false positives equals $1 - precision$, and that the percentage of false negatives equals $1 - recall$.

classifier	*helpful*	*neutral*	*harmful*
simple	67	333	44
strict-red	113	295	36
relaxed-red	211	233	0

Table 5: Effectiveness of the classifiers at focusing the programmer's attention on failure-inducing changes.

serve that both the *strict-green* and *relaxed-green* classifiers have a precision of 100%. In other words, we find that, in this case study, each classifier has the desirable property that changes classified as *Green* are never failure-inducing. Moreover, for *Green* changes, the *relaxed-green* classifiers produce a recall of 19.3% versus 14.5% for the *strict-green* classifiers. This suggests that the *relaxed-green* classifiers are the most successful at classifying non-failure-inducing changes as *Green*.[13] Consequently, the *relaxed-green* classifiers are clearly preferable over the *strict-green* ones. Combining our conclusions about the classification of *Red* and *Green* changes, we conclude that the *relaxed-red*/*relaxed-green* classifier produces the best results.

Per-Test Evaluation. As a final step in this case study, we measure how often change classification helps 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 we compare to is the uncolored set of affecting changes as calculated by *Chianti*. Table 5 shows how often the *simple*, *strict-red* and *relaxed-red* classifiers are helpful at focusing the programmer's attention on the failure-inducing changes. The *helpful* column counts the number of tests where all failure-inducing changes are colored *Red*, and some of the other affecting changes for the test are *Yellow*. In other words, *helpful* cases are those where change classification provides more accurate information than the (uncolored) affecting changes and thus helps focus the programmer's attention. The *neutral* column counts the number of tests for which all affecting changes are *Red and* the number of tests for which all affecting changes are *Yellow*. In other words, *neutral* cases are those where change classification is neither helpful nor harmful. The *harmful* column counts the number of tests for which some failure-inducing changes are colored *Yellow*, and where some non-failure-inducing changes are colored *Red*. Hence, *harmful* cases are those where the coloring actively points the programmer at the wrong changes. The results of Table 5 shows that the *relaxed-red* classifiers are successful at focusing programmer attention in 47.5% (211/444) of the cases, while doing no harm in the remaining cases. The *strict-red* and *simple* classifiers are not only less effective, but they also produce harmful results in a significant percentage of the cases, thus confirming our conclusion that the *relaxed-red* classifiers should be preferred.

Committable changes. For the student data we also counted the committable changes in the student code, according to Definition 3.4. Of the initial 6624 changes, 53.6% (3553) are committable. This shows that change classification is likely to be useful in collaborative development scenarios, by enabling developers to expose changes to team members more quickly. Note that for this case study we were dealing with tests that are relatively high-level, resulting in many dependences between affecting changes. For systems and test suites that are more modular, we expect an even higher percentage of committable changes.

[13] While a recall of 19.3% is still low, this is an artifact of our evaluation data (few changes, significant number of them failure-inducing) and our desire to be conservative by never classifying a change affecting a worsening test as *Green*.

4.2 Case Study 2: Daikon

Daikon [6] 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. 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 *relaxed-red* classifiers failed to focus on these changes because they 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 change to be a **CM** change to method `daikon.diff.Diff.shouldAdd()`. Again, the *relaxed-red* classifiers were not useful because they 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 6093 changes of which 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 *strict-red*/*relaxed-green* classifier.

5278 of the 6093 changes (86.6%) were classified as committable, according to Definition 3.4. In this case, a high coupling existed between tests and changes (one change affected 36/40 tests), and no covered (i.e., non-*Gray*) changes were found to be committable due to the conservative treatment of semantic dependences.

4.3 Assessment

While it seems contradictory that the case studies involving student data and Daikon suggest that different classifiers 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 [27], inputs [30], thread schedules [3], or program states [29, 4] 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 whether or not a change is failure-inducing by observing the effect of its presence or absence in two program executions. 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 runtime. 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. [18] 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. [8] 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. [9] 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.

Dallmeier et al. [5] 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 [9], and that comparing sequences of method calls *on the same object* is an even better predictor.

Liblit et al. [13, 14] 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 [13] applies in situations with a single bug, whereas that of [14] can identify separate causes in the presence of multiple bugs.

Change Impact Analysis. We previously presented the conceptual framework [19] of our change impact analysis, and its expansion to the full Java language with empirical validation [17]. 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 [1, 12, 10, 26], dynamic analysis [11] or, like our analysis, on a combination of the two [15]. Recent work on change impact analysis includes the *PathImpact* algorithm by Law and Rothermel [11], 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. [15], which combines the use of a forward static slice [24] 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 [16].

Continuous Testing and Test Factoring. Saff and Ernst [20, 23] 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 [22], these results are confirmed in a comparative case study with student developers.

The same authors introduce *test factoring* [21], 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 continuous testing and test factoring by

directly reducing fix time.

6. CONCLUSIONS

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 described how change classification can be used to determine a set of changes that can be released safely to a version control repository, even in cases where the programmer's local workspace contains failing tests. Third, we implemented these change classification techniques in *JUnit/CIA*, an extension of the *JUnit* component of *Eclipse*, and conducted two case studies. In the student programs case study, the *relaxed-red/relaxed-green* classifier focused programmer attention on failure-inducing changes in 47.5% of the version pairs, without providing misleading information in the remaining cases. In the same case study, 53.6% of the changes were found to be committable, in spite of our coarse approximation of semantic dependences. In the *Daikon* case study, another classifier (*strict-red/relaxed-green*) was the most effective due to the different nature of the data involved. While these results are promising, it is clear that more data and/or a user study are needed for a full validation of the approach.

Acknowledgements. This research was supported by NSF grant CCR-0204410 and, in part, by IBM Research.

7. REFERENCES

[1] 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.

[2] CHESLEY, O., REN, X., AND RYDER, B. G. Crisp: A debugging tool for Java programs. In *Proc. of the Int. Conf. on Software Maintentance* (September 2005).

[3] 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.

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

[5] 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).

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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).

[13] 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.

[14] 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).

[15] 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).

[16] 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.

[17] 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.

[18] 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.

[19] 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).

[20] 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.

[21] 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.

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

[23] 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).

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

APPENDIX

This appendix contains detailed information for each of the five analyzers and each individual of all 98 versions containing failure inducing changes.

Legend: '#' indicates absolute numbers, '%' a percentage. 'VS' identifies the version pair, '|Δ|' the number of changes total, 'FIC' failure-inducing changes, 'FP' false positives and 'FN' false negatives. 'Prec.' is abreviated for precision.

A. *relaxed-red/relaxed-green*

Student	VS	|Δ|	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
3474	109	2	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
13455	104	55	2	11	5	0	9	0	81.8	0.0	0.0	100.0	100.0	18.2
15753	126	15	3	9	0	0	6	0	66.7	0.0	0.0	100.0	100.0	33.3
15828	113	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16560	104	8	2	0	2	3	0	2	0.0	100.0	50.0	100.0	0.0	100.0
16560	119	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16560	122	3	3	3	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16788	107	29	2	0	3	5	0	2	0.0	100.0	18.5	100.0	0.0	100.0
16788	109	7	1	2	3	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
16788	130	19	1	1	1	6	0	0	0.0	0.0	33.3	100.0	100.0	100.0
16788	135	5	1	0	1	2	0	1	0.0	100.0	50.0	100.0	0.0	100.0
16830	108	12	1	0	6	4	0	1	0.0	100.0	36.4	100.0	0.0	100.0
16830	110	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
16830	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16830	116	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16830	120	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	155	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	156	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	160	5	1	0	1	4	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16836	180	5	2	4	1	0	3	1	75.0	50.0	0.0	100.0	50.0	25.0
16899	113	3	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16953	115	21	2	9	4	0	7	0	77.8	0.0	0.0	100.0	100.0	22.2
16953	116	2	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16953	123	19	1	7	3	0	6	0	85.7	0.0	0.0	100.0	100.0	14.3
17082	108	6	1	2	0	1	1	0	50.0	0.0	20.0	100.0	100.0	50.0
17094	107	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	111	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	113	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	115	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	118	33	1	0	1	8	0	1	0.0	100.0	25.0	100.0	0.0	100.0
17094	120	17	3	0	11	0	0	3	0.0	100.0	0.0	100.0	0.0	100.0
17094	121	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	122	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	123	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	125	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	129	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	132	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17136	112	11	2	0	4	4	0	2	0.0	100.0	44.4	100.0	0.0	100.0
17283	110	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	120	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	148	3	1	3	0	0	2	0	66.7	0.0	0.0	100.0	100.0	33.3
17448	117	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	121	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	129	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	141	14	6	1	9	0	0	5	0.0	83.3	0.0	100.0	16.7	100.0
17448	144	3	2	1	1	0	1	2	100.0	100.0	0.0	100.0	0.0	0.0
17661	108	16	1	4	0	0	3	0	75.0	0.0	0.0	100.0	100.0	25.0
17661	119	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17661	125	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17661	133	10	1	1	4	1	0	0	0.0	0.0	11.1	100.0	100.0	100.0
17661	134	10	1	2	2	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0

Student	VS	$\|\Delta\|$	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
17661	139	10	2	3	0	1	1	0	33.3	0.0	12.5	100.0	100.0	66.7
17661	140	9	1	0	4	3	0	1	0.0	100.0	37.5	100.0	0.0	100.0
17661	142	2	1	0	1	1	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17661	150	2	1	0	1	1	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17823	115	4	1	3	0	0	2	0	66.7	0.0	0.0	100.0	100.0	33.3
17841	114	13	4	4	2	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17841	116	10	4	4	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17841	118	10	4	4	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17895	106	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17895	114	257	1	0	2	8	0	1	0.0	100.0	3.1	100.0	0.0	100.0
17895	121	6	1	0	2	2	0	1	0.0	100.0	40.0	100.0	0.0	100.0
17895	132	2	1	1	0	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17895	134	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18021	113	83	1	20	1	4	19	0	95.0	0.0	4.9	100.0	100.0	5.0
18021	116	11	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18093	117	3	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18123	104	28	1	0	1	13	0	1	0.0	100.0	48.1	100.0	0.0	100.0
18123	121	5	1	1	1	3	0	0	0.0	0.0	75.0	100.0	100.0	100.0
18123	122	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18123	123	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18123	126	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18123	136	5	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18234	106	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18312	105	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18312	116	11	2	7	0	0	5	0	71.4	0.0	0.0	100.0	100.0	28.6
18312	120	2	2	2	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18312	126	6	3	3	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
18717	111	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18717	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18717	128	138	2	1	1	5	0	1	0.0	50.0	3.7	100.0	50.0	100.0
18717	129	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18717	138	8	1	1	0	4	0	0	0.0	0.0	57.1	100.0	100.0	100.0
18717	146	11	2	1	2	2	0	1	0.0	50.0	22.2	100.0	50.0	100.0
21228	106	126	1	0	7	27	0	1	0.0	100.0	21.6	100.0	0.0	100.0
21435	124	26	1	1	1	12	0	0	0.0	0.0	48.0	100.0	100.0	100.0
21477	115	110	2	0	19	12	0	2	0.0	100.0	11.1	100.0	0.0	100.0
21477	145	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	150	5	1	1	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
21477	180	16	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	185	11	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
21477	187	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	193	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	195	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21879	108	17	1	8	0	1	7	0	87.5	0.0	6.3	100.0	100.0	12.5
21879	109	6	1	1	0	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
21879	131	3	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0

B. *strict-red/relaxed-green*

Student	VS	$\|\Delta\|$	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
3474	109	2	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
13455	104	55	2	8	8	0	8	2	100.0	100.0	0.0	100.0	0.0	0.0
15753	126	15	3	0	9	0	0	3	0.0	100.0	0.0	100.0	0.0	100.0
15828	113	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16560	104	8	2	0	2	3	0	2	0.0	100.0	50.0	100.0	0.0	100.0
16560	119	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16560	122	3	3	0	3	0	0	3	0.0	100.0	100.0	100.0	0.0	100.0
16788	107	29	2	0	3	5	0	2	0.0	100.0	18.5	100.0	0.0	100.0
16788	109	7	1	2	3	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0

Student	VS	$\|\Delta\|$	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
16788	130	19	1	0	2	6	0	1	0.0	100.0	33.3	100.0	0.0	100.0
16788	135	5	1	0	1	2	0	1	0.0	100.0	50.0	100.0	0.0	100.0
16830	108	12	1	0	6	4	0	1	0.0	100.0	36.4	100.0	0.0	100.0
16830	110	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
16830	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16830	116	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16830	120	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16836	155	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	156	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	160	5	1	0	1	4	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16836	180	5	2	4	1	0	3	1	75.0	50.0	0.0	100.0	50.0	25.0
16899	113	3	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16953	115	21	2	9	4	0	7	0	77.8	0.0	0.0	100.0	100.0	22.2
16953	116	2	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16953	123	19	1	0	10	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17082	108	6	1	0	2	1	0	1	0.0	100.0	20.0	100.0	0.0	100.0
17094	107	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	111	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	113	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	115	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	118	33	1	0	1	8	0	1	0.0	100.0	25.0	100.0	0.0	100.0
17094	120	17	3	0	11	0	0	3	0.0	100.0	0.0	100.0	0.0	100.0
17094	121	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	122	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	123	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	125	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	129	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	132	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17136	112	11	2	0	4	4	0	2	0.0	100.0	44.4	100.0	0.0	100.0
17283	110	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	120	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	148	3	1	0	3	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17448	117	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	121	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	129	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	141	14	6	1	9	0	0	5	0.0	83.3	0.0	100.0	16.7	100.0
17448	144	3	2	1	1	0	1	2	100.0	100.0	0.0	100.0	0.0	0.0
17661	108	16	1	2	2	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
17661	119	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17661	125	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17661	133	10	1	1	4	1	0	0	0.0	0.0	11.1	100.0	100.0	100.0
17661	134	10	1	0	4	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17661	139	10	2	2	1	1	0	0	0.0	0.0	12.5	100.0	100.0	100.0
17661	140	9	1	0	4	3	0	1	0.0	100.0	37.5	100.0	0.0	100.0
17661	142	2	1	0	1	1	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17661	150	2	1	0	1	1	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17823	115	4	1	0	3	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17841	114	13	4	4	2	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17841	116	10	4	4	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17841	118	10	4	4	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17895	106	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17895	114	257	1	0	2	8	0	1	0.0	100.0	3.1	100.0	0.0	100.0
17895	121	6	1	0	2	2	0	1	0.0	100.0	40.0	100.0	0.0	100.0
17895	132	2	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17895	134	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18021	113	83	1	20	1	4	19	0	95.0	0.0	4.9	100.0	100.0	5.0
18021	116	11	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18093	117	3	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18123	104	28	1	0	1	13	0	1	0.0	100.0	48.1	100.0	0.0	100.0
18123	121	5	1	0	2	3	0	1	0.0	100.0	75.0	100.0	0.0	100.0

Student	VS	|Δ|	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
18123	122	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
18123	123	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
18123	126	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18123	136	5	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18234	106	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
18312	105	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18312	116	11	2	2	5	0	1	1	50.0	50.0	0.0	100.0	50.0	50.0
18312	120	2	2	1	1	0	0	1	0.0	50.0	100.0	100.0	50.0	100.0
18312	126	6	3	3	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
18717	111	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18717	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18717	128	138	2	1	1	5	0	1	0.0	50.0	3.7	100.0	50.0	100.0
18717	129	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18717	138	8	1	0	1	4	0	1	0.0	100.0	57.1	100.0	0.0	100.0
18717	146	11	2	0	3	2	0	2	0.0	100.0	22.2	100.0	0.0	100.0
21228	106	126	1	0	7	27	0	1	0.0	100.0	21.6	100.0	0.0	100.0
21435	124	26	1	0	2	12	0	1	0.0	100.0	48.0	100.0	0.0	100.0
21477	115	110	2	0	19	12	0	2	0.0	100.0	11.1	100.0	0.0	100.0
21477	145	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	150	5	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	180	16	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	185	11	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	187	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
21477	193	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	195	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
21879	108	17	1	1	7	1	0	0	0.0	0.0	6.3	100.0	100.0	100.0
21879	109	6	1	1	0	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
21879	131	3	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0

C. *relaxed-red/strict-green*

Student	VS	|Δ|	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
3474	109	2	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
13455	104	55	2	11	5	0	9	0	81.8	0.0	0.0	100.0	100.0	18.2
15753	126	15	3	9	0	0	6	0	66.7	0.0	0.0	100.0	100.0	33.3
15828	113	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16560	104	8	2	0	2	3	0	2	0.0	100.0	50.0	100.0	0.0	100.0
16560	119	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16560	122	3	3	3	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16788	107	29	2	0	3	5	0	2	0.0	100.0	18.5	100.0	0.0	100.0
16788	109	7	1	2	3	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
16788	130	19	1	1	1	6	0	0	0.0	0.0	33.3	100.0	100.0	100.0
16788	135	5	1	0	1	2	0	1	0.0	100.0	50.0	100.0	0.0	100.0
16830	108	12	1	0	10	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16830	110	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
16830	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16830	116	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16830	120	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	155	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	156	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	160	5	1	0	1	4	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16836	180	5	2	4	1	0	3	1	75.0	50.0	0.0	100.0	50.0	25.0
16899	113	3	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16953	115	21	2	9	4	0	7	0	77.8	0.0	0.0	100.0	100.0	22.2
16953	116	2	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16953	123	19	1	7	3	0	6	0	85.7	0.0	0.0	100.0	100.0	14.3
17082	108	6	1	2	0	1	1	0	50.0	0.0	20.0	100.0	100.0	50.0
17094	107	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	111	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	113	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0

Student	VS	$\|\Delta\|$	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
17094	115	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	118	33	1	0	9	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17094	120	17	3	0	11	0	0	3	0.0	100.0	0.0	100.0	0.0	100.0
17094	121	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	122	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	123	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	125	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	129	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	132	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17136	112	11	2	0	4	4	0	2	0.0	100.0	44.4	100.0	0.0	100.0
17283	110	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	120	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	148	3	1	3	0	0	2	0	66.7	0.0	0.0	100.0	100.0	33.3
17448	117	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	121	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	129	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	141	14	6	1	9	0	0	5	0.0	83.3	0.0	100.0	16.7	100.0
17448	144	3	2	1	1	0	1	2	100.0	100.0	0.0	100.0	0.0	0.0
17661	108	16	1	4	0	0	3	0	75.0	0.0	0.0	100.0	100.0	25.0
17661	119	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17661	125	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17661	133	10	1	1	5	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17661	134	10	1	2	2	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
17661	139	10	2	3	0	1	1	0	33.3	0.0	12.5	100.0	100.0	66.7
17661	140	9	1	0	7	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17661	142	2	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17661	150	2	1	0	1	1	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17823	115	4	1	3	0	0	2	0	66.7	0.0	0.0	100.0	100.0	33.3
17841	114	13	4	4	2	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17841	116	10	4	4	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17841	118	10	4	4	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17895	106	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17895	114	257	1	0	2	8	0	1	0.0	100.0	3.1	100.0	0.0	100.0
17895	121	6	1	0	2	2	0	1	0.0	100.0	40.0	100.0	0.0	100.0
17895	132	2	1	1	0	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17895	134	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18021	113	83	1	20	4	1	19	0	95.0	0.0	1.2	100.0	100.0	5.0
18021	116	11	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18093	117	3	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18123	104	28	1	0	14	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18123	121	5	1	1	1	3	0	0	0.0	0.0	75.0	100.0	100.0	100.0
18123	122	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18123	123	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18123	126	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18123	136	5	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18234	106	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18312	105	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18312	116	11	2	7	0	0	5	0	71.4	0.0	0.0	100.0	100.0	28.6
18312	120	2	2	2	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18312	126	6	3	3	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
18717	111	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18717	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18717	128	138	2	1	4	2	0	1	0.0	50.0	1.5	100.0	50.0	100.0
18717	129	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18717	138	8	1	1	0	4	0	0	0.0	0.0	57.1	100.0	100.0	100.0
18717	146	11	2	1	2	2	0	1	0.0	50.0	22.2	100.0	50.0	100.0
21228	106	126	1	0	33	1	0	1	0.0	100.0	.8	100.0	0.0	100.0
21435	124	26	1	1	1	12	0	0	0.0	0.0	48.0	100.0	100.0	100.0
21477	115	110	2	0	31	0	0	2	0.0	100.0	0.0	100.0	0.0	100.0
21477	145	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0

Student	VS	$\|\Delta\|$	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
21477	150	5	1	1	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
21477	180	16	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	185	11	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
21477	187	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	193	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	195	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21879	108	17	1	8	0	1	7	0	87.5	0.0	6.3	100.0	100.0	12.5
21879	109	6	1	1	0	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
21879	131	3	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0

D. *strict-red/strict-green*

Student	VS	$\|\Delta\|$	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
3474	109	2	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
13455	104	55	2	8	8	0	8	2	100.0	100.0	0.0	100.0	0.0	0.0
15753	126	15	3	0	9	0	0	3	0.0	100.0	0.0	100.0	0.0	100.0
15828	113	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16560	104	8	2	0	2	3	0	2	0.0	100.0	50.0	100.0	0.0	100.0
16560	119	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16560	122	3	3	0	3	0	0	3	0.0	100.0	100.0	100.0	0.0	100.0
16788	107	29	2	0	3	5	0	2	0.0	100.0	18.5	100.0	0.0	100.0
16788	109	7	1	2	3	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
16788	130	19	1	0	2	6	0	1	0.0	100.0	33.3	100.0	0.0	100.0
16788	135	5	1	0	1	2	0	1	0.0	100.0	50.0	100.0	0.0	100.0
16830	108	12	1	0	10	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16830	110	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
16830	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16830	116	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16830	120	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16836	155	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	156	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	160	5	1	0	1	4	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16836	180	5	2	4	1	0	3	1	75.0	50.0	0.0	100.0	50.0	25.0
16899	113	3	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16953	115	21	2	9	4	0	7	0	77.8	0.0	0.0	100.0	100.0	22.2
16953	116	2	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16953	123	19	1	0	10	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17082	108	6	1	0	2	1	0	1	0.0	100.0	20.0	100.0	0.0	100.0
17094	107	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	111	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	113	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	115	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	118	33	1	0	9	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17094	120	17	3	0	11	0	0	3	0.0	100.0	0.0	100.0	0.0	100.0
17094	121	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	122	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	123	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	125	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	129	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	132	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17136	112	11	2	0	4	4	0	2	0.0	100.0	44.4	100.0	0.0	100.0
17283	110	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	120	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	148	3	1	0	3	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17448	117	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	121	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	129	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	141	14	6	1	9	0	0	5	0.0	83.3	0.0	100.0	16.7	100.0
17448	144	3	2	1	1	0	1	2	100.0	100.0	0.0	100.0	0.0	0.0

Student	VS	$\|\Delta\|$	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
17661	108	16	1	2	2	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
17661	119	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17661	125	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17661	133	10	1	1	5	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17661	134	10	1	0	4	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17661	139	10	2	2	1	1	0	0	0.0	0.0	12.5	100.0	100.0	100.0
17661	140	9	1	0	7	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17661	142	2	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17661	150	2	1	0	1	1	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17823	115	4	1	0	3	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17841	114	13	4	4	2	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17841	116	10	4	4	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17841	118	10	4	4	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
17895	106	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17895	114	257	1	0	2	8	0	1	0.0	100.0	3.1	100.0	0.0	100.0
17895	121	6	1	0	2	2	0	1	0.0	100.0	40.0	100.0	0.0	100.0
17895	132	2	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17895	134	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18021	113	83	1	20	4	1	19	0	95.0	0.0	1.2	100.0	100.0	5.0
18021	116	11	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18093	117	3	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18123	104	28	1	0	14	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18123	121	5	1	0	2	3	0	1	0.0	100.0	75.0	100.0	0.0	100.0
18123	122	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
18123	123	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
18123	126	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18123	136	5	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18234	106	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
18312	105	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18312	116	11	2	2	5	0	1	1	50.0	50.0	0.0	100.0	50.0	50.0
18312	120	2	2	1	1	0	0	1	0.0	50.0	100.0	100.0	50.0	100.0
18312	126	6	3	3	1	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
18717	111	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18717	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18717	128	138	2	1	4	2	0	1	0.0	50.0	1.5	100.0	50.0	100.0
18717	129	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18717	138	8	1	0	1	4	0	1	0.0	100.0	57.1	100.0	0.0	100.0
18717	146	11	2	0	3	2	0	2	0.0	100.0	22.2	100.0	0.0	100.0
21228	106	126	1	0	33	1	0	1	0.0	100.0	.8	100.0	0.0	100.0
21435	124	26	1	0	2	12	0	1	0.0	100.0	48.0	100.0	0.0	100.0
21477	115	110	2	0	31	0	0	2	0.0	100.0	0.0	100.0	0.0	100.0
21477	145	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	150	5	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	180	16	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	185	11	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	187	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
21477	193	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	195	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
21879	108	17	1	1	7	1	0	0	0.0	0.0	6.3	100.0	100.0	100.0
21879	109	6	1	1	0	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
21879	131	3	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0

E. *simple*CLASSIFIER

Student	VS	$\|\Delta\|$	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
3474	109	2	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
13455	104	55	2	12	4	0	12	2	100.0	100.0	0.0	100.0	0.0	0.0
15753	126	15	3	0	9	0	0	3	0.0	100.0	0.0	100.0	0.0	100.0
15828	113	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16560	104	8	2	0	2	3	0	2	0.0	100.0	50.0	100.0	0.0	100.0

Student	VS	$\|\Delta\|$	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
16560	119	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16560	122	3	3	0	3	0	0	3	0.0	100.0	100.0	100.0	0.0	100.0
16788	107	29	2	7	1	0	7	2	100.0	100.0	0.0	100.0	0.0	0.0
16788	109	7	1	2	3	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
16788	130	19	1	0	2	6	0	1	0.0	100.0	33.3	100.0	0.0	100.0
16788	135	5	1	0	1	2	0	1	0.0	100.0	50.0	100.0	0.0	100.0
16830	108	12	1	0	10	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16830	110	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
16830	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16830	116	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16830	120	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
16836	155	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	156	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
16836	160	5	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16836	180	5	2	4	1	0	3	1	75.0	50.0	0.0	100.0	50.0	25.0
16899	113	3	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16953	115	21	2	13	0	0	11	0	84.6	0.0	0.0	100.0	100.0	15.4
16953	116	2	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
16953	123	19	1	0	10	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17082	108	6	1	0	2	1	0	1	0.0	100.0	20.0	100.0	0.0	100.0
17094	107	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	111	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	113	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	115	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	118	33	1	9	0	0	9	1	100.0	100.0	0.0	100.0	0.0	0.0
17094	120	17	3	0	11	0	0	3	0.0	100.0	0.0	100.0	0.0	100.0
17094	121	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	122	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	123	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	125	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17094	129	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17094	132	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17136	112	11	2	0	4	4	0	2	0.0	100.0	44.4	100.0	0.0	100.0
17283	110	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	120	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17283	148	3	1	0	3	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17448	117	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	121	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	129	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17448	141	14	6	4	6	0	3	5	75.0	83.3	0.0	100.0	16.7	25.0
17448	144	3	2	1	1	0	1	2	100.0	100.0	0.0	100.0	0.0	0.0
17661	108	16	1	2	2	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
17661	119	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17661	125	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
17661	133	10	1	3	3	0	2	0	66.7	0.0	0.0	100.0	100.0	33.3
17661	134	10	1	2	2	0	2	1	100.0	100.0	0.0	100.0	0.0	0.0
17661	139	10	2	2	1	1	0	0	0.0	0.0	12.5	100.0	100.0	100.0
17661	140	9	1	3	4	0	3	1	100.0	100.0	0.0	100.0	0.0	0.0
17661	142	2	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17661	150	2	1	0	1	1	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17823	115	4	1	0	3	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17841	114	13	4	6	0	0	2	0	33.3	0.0	0.0	100.0	100.0	66.7
17841	116	10	4	5	0	0	1	0	20.0	0.0	0.0	100.0	100.0	80.0
17841	118	10	4	5	0	0	1	0	20.0	0.0	0.0	100.0	100.0	80.0
17895	106	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
17895	114	257	1	0	10	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17895	121	6	1	0	2	2	0	1	0.0	100.0	40.0	100.0	0.0	100.0
17895	132	2	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
17895	134	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18021	113	83	1	21	3	1	20	0	95.2	0.0	1.2	100.0	100.0	4.8

Student	VS	$\|\Delta\|$	#FIC	*#Red*	*#Yellow*	*#Green*	#FP	#FN	%FP	%FN	Recall *Green*	Prec. *Green*	Recall *Red*	Prec. *Red*
18021	116	11	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18093	117	3	1	0	1	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18123	104	28	1	0	14	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18123	121	5	1	0	2	3	0	1	0.0	100.0	75.0	100.0	0.0	100.0
18123	122	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
18123	123	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
18123	126	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18123	136	5	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
18234	106	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
18312	105	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18312	116	11	2	2	5	0	1	1	50.0	50.0	0.0	100.0	50.0	50.0
18312	120	2	2	1	1	0	0	1	0.0	50.0	100.0	100.0	50.0	100.0
18312	126	6	3	4	0	0	1	0	25.0	0.0	0.0	100.0	100.0	75.0
18717	111	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18717	114	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
18717	128	138	2	3	4	0	2	1	66.7	50.0	0.0	100.0	50.0	33.3
18717	129	2	1	2	0	0	1	0	50.0	0.0	0.0	100.0	100.0	50.0
18717	138	8	1	0	1	4	0	1	0.0	100.0	57.1	100.0	0.0	100.0
18717	146	11	2	0	3	2	0	2	0.0	100.0	22.2	100.0	0.0	100.0
21228	106	126	1	0	33	1	0	1	0.0	100.0	.8	100.0	0.0	100.0
21435	124	26	1	0	2	12	0	1	0.0	100.0	48.0	100.0	0.0	100.0
21477	115	110	2	0	31	0	0	2	0.0	100.0	0.0	100.0	0.0	100.0
21477	145	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	150	5	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	180	16	1	0	5	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	185	11	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0
21477	187	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
21477	193	1	1	1	0	0	0	0	0.0	0.0	100.0	100.0	100.0	100.0
21477	195	1	1	0	1	0	0	1	0.0	100.0	100.0	100.0	0.0	100.0
21879	108	17	1	1	7	1	0	0	0.0	0.0	6.3	100.0	100.0	100.0
21879	109	6	1	1	0	0	0	0	0.0	0.0	0.0	100.0	100.0	100.0
21879	131	3	1	0	2	0	0	1	0.0	100.0	0.0	100.0	0.0	100.0