Safe-Commit Analysis to Facilitate Team Software Development

Jan Wloka[*], Barbara Ryder[†], Frank Tip[‡] and Xiaoxia Ren[*]
[*] Dept. of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
{jwloka, xren}@cs.rutgers.edu
[†] Dept. of Computer Science, Virginia Tech, Blacksburg, VA 24061, USA
ryder@cs.vt.edu
[‡] IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
ftip@us.ibm.com

Abstract

Software development teams exchange source code in shared repositories. These repositories are kept consistent by having developers follow a commit policy, such as "Program edits can be committed only if all available tests succeed." Such policies may result in long intervals between commits, increasing the likelihood of duplicative development and merge conflicts. Furthermore, commit policies are generally not automatically enforceable.

We present a program analysis to identify committable changes that can be released early, without causing failures of existing tests, even in the presence of failing tests in a developer's local workspace. The algorithm can support relaxed commit policies that allow early release of changes, reducing the potential for merge conflicts. In experiments using several versions of a non-trivial software system with failing tests, 3 newly enabled commit policies were shown to allow a significant percentage of changes to be committed.

1 Introduction

Software development of large systems today is a highly collaborative process where teamwork is essential. Team development reduces the time to market, but the cost of coordination problems caused by duplicative and conflicting edits in a code base can be nontrivial [17]. Although it is customary to assign clear responsibilities for each module, team members may be affected unavoidably when their changes conflict with the changes made by others.

Modern revision control systems such as CVS [4] and Subversion [18] can automatically resolve *direct merge conflicts* that arise when multiple developers concurrently access the same file. However, this conflict resolution is limited in several ways:

- Manual conflict resolution is needed when the edits involve overlapping text regions.
- Current revision control systems are unable to detect *indirect merge conflicts* that arise when the changes made by different developers on *different* files adversely impact each other.
- Most importantly, the detection and resolution of conflicts in current revision control systems is based on a textual analysis, and unexpected interactions between changes may cause erroneous program behavior, even in cases where no conflicts are reported.

When problems arise, they often manifest themselves as test failures experienced by other team members. Such test failures are notoriously difficult to debug for developers because the problem was not caused by their own changes.

Therefore, the process of committing changes to a shared repository is typically governed by a *commit policy* that aims to minimize merge conflicts, eliminate build problems and avoid test outcome degradations. A policy imposed by the project management usually consists of a small number of informally stated guidelines that developers are encouraged to follow. For example, many development teams follow the "Commit early and commit often" rule (see, e.g., [3, 11]), in order to avoid long time intervals between commits that may lead to duplicative development and merge conflicts. To preserve code quality, the commit policy followed by the KDE team [14] includes rules such as "Never commit code that doesn't compile" and "Test your changes before committing". The latter is commonly interpreted to mean that all tests in a developer's local workspace must pass before changes can be committed; this corresponds to the *conservative* policy: "Do not commit changes in the presence of failing tests in the local workspace". Unfortunately, this requirement also will generally increase the

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

time intervals between commits, at odds with the "Commit early and commit often" rule.

In addition, the *conservative* policy may be overly restrictive. For example, test failures that already occur in the original version of the program may be completely unrelated to the developer's changes, and fixing them may require very different knowledge of the application[1]. Furthermore, several widely used development methodologies such as extreme programming [2] and test-driven development [1] advocate that tests be written before the tested functionality is implemented. Such tests initially fail until the functionality is implemented, and the *conservative* policy blocks such tests from being released, thereby preventing developers from collaborating on test development.

We conclude from the above discussion that it is desirable to use a more relaxed commit policy that allows parts of an evolving program to be committed before the entire implementation of a feature is complete, provided that the released changes do not introduce additional test failures. Our research focuses on using program analysis to define practical relaxed commit policies that prevent the introduction of new test failures, while allowing early commits of changes. Thus, the analysis determines a set of safe *committable changes* that can be exposed to other team members. The analysis is based on an existing change impact technique [20] that compares successive versions of a software system and expresses their difference as a set of coarse-grained *atomic changes*. Each test in a test suite, represented by its dynamic call graph, is then correlated with the changes that may affect its outcome. We present *Safe-commit*, an algorithm that computes a committable subset of the changes in the edit using this *change-test correlation*.

In different development strategies, changes can be considered to be not safely committable for varying reasons. Using the change-test correlation, a change can be unsafe because it is covered by a failing test, covered by a worsening test (that passed before), or not covered by any test. We show how *Safe-commit* can be used to implement 3 commit policies of varying degrees of "strictness". The *Restrictive* policy allows only the release of changes that are "successfully tested", that is, they only impact tests that pass in the edited program version. This policy is useful in situations where the release of changes needs to be controlled tightly, e.g., when a major release is imminent and only bug fixes are permissable. The *Moderate* policy permits the release of changes that are tested, provided that no outcome of an existing test degrades. It is useful, e.g., for test-driven development, since every change has to be tested but failing tests can be added to the suite. Finally, the *Permissive* policy relaxes the *Moderate* policy by additionally allowing the release of untested changes. It is probably used by many projects as the default policy. Either of the *Moderate* or *Permissive* policies can be used instead of the conservative policy without compromising the code base in the repository.

The contributions of this paper are threefold:

- *Safe-commit, an algorithm to calculate a set of safely committable changes*. Releasing these changes is guaranteed not to cause the failure of any existing test. To our knowledge, this is the *first semantic analysis for calculating committable changes*;
- A *prototype implementation* of *Safe-commit* in JUnitMX, a plug-in that seamlessly extends the JUnit support in Eclipse, and shows developers the set of committable changes after running their test suite; and
- A *preliminary evaluation* of three new commit policies that were implemented using *Safe-commit* on several versions of *Daikon* [9]. In this experiment, an average of 4.6%, 31.4%, and 99.5% of all atomic changes were identified as committable according to the *Restrictive*, *Moderate*, and *Permissive* commit policies, respectively.

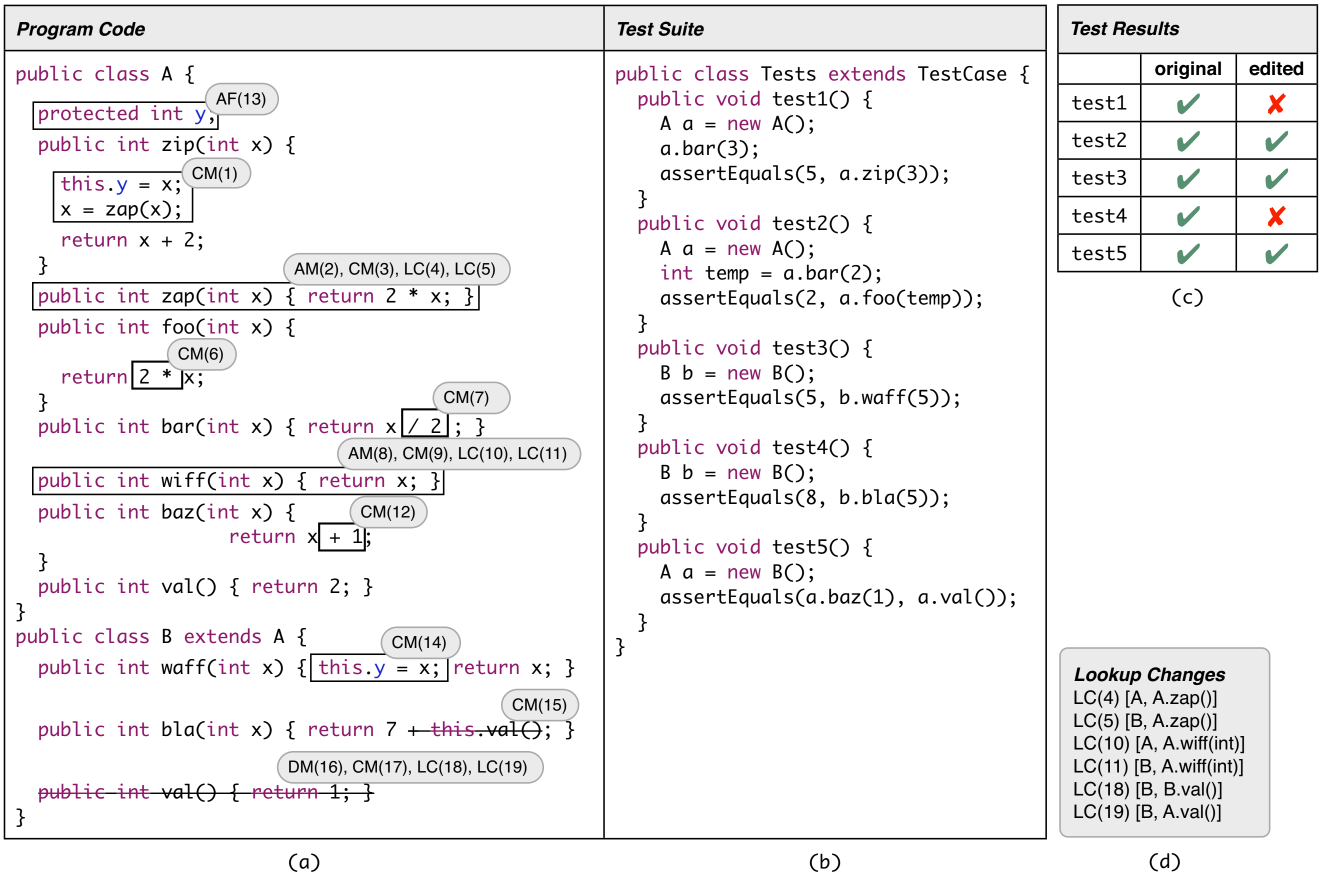
We also report on an experiment where *Safe-commit* is applied to *public releases* of *JMeter* [8]. Surprisingly, *Safe-commit* was capable of identifying a nontrivial number of committable changes, despite the huge number of changes that separates these releases.

2 Motivating Example

The program shown in Figure 1 will serve as a running example throughout the paper to illustrate the algorithm. Part (a) of the figure shows the program itself, and part (b) shows the associated test suite. Since we will use two versions of this program to illustrate our approach, program changes are indicated with boxes (for additions) and with ~~strikeout~~ font (for deletions). In other words, the original program contains none of the boxed code fragments and all of the code fragments in strikeout font, and the edited program version is constructed by adding all boxed code fragments and removing all fragments in strikeout font. In Figure 1, gray labels are used to indicate various kinds of changes (e.g., AM changes correspond to added methods). These annotations are used to illustrate our analysis approach and will be explained in Section 3.

The example program consists of two classes `A` and `B` and a test suite of five tests. In the original program version,

[1] For example, if a test t fails in both the original and the edited version of a program, then the failure of t in the edited program may be caused by the changes, or it may be due to the same reason that caused t's failure in the original version. Determining why t fails in the edited program is beyond the scope of this paper; therefore, we have concentrated on developing an analysis that guarantees that there are no additional tests that fail after committing changes.


AF(13)
CM(1)
AM(2), CM(3), LC(4), LC(5)
CM(6)
CM(7)
AM(8), CM(9), LC(10), LC(11)
CM(12)
CM(14)
CM(15)
DM(16), CM(17), LC(18), LC(19)
(a)
(b)
(c)
(d)

Program Code

```
public class A {
  protected int y;
  public int zip(int x) {
    this.y = x;
    x = zap(x);
    return x + 2;
  }
  public int zap(int x) { return 2 * x; }
  public int foo(int x) {
    return 2 * x;
  }
  public int bar(int x) { return x / 2; }
  public int wiff(int x) { return x; }
  public int baz(int x) {
                 return x + 1;
  }
  public int val() { return 2; }
}
public class B extends A {
  public int waff(int x) { this.y = x; return x; }

  public int bla(int x) { return 7 ~~+ this.val()~~; }

  ~~public int val() { return 1; }~~
}
```

Test Suite

```
public class Tests extends TestCase {
  public void test1() {
    A a = new A();
    a.bar(3);
    assertEquals(5, a.zip(3));
  }
  public void test2() {
    A a = new A();
    int temp = a.bar(2);
    assertEquals(2, a.foo(temp));
  }
  public void test3() {
    B b = new B();
    assertEquals(5, b.waff(5));
  }
  public void test4() {
    B b = new B();
    assertEquals(8, b.bla(5));
  }
  public void test5() {
    A a = new B();
    assertEquals(a.baz(1), a.val());
  }
}
```

Test Results

	original	edited
test1	✓	✗
test2	✓	✓
test3	✓	✓
test4	✓	✗
test5	✓	✓

Lookup Changes
LC(4) [A, A.zap()]
LC(5) [B, A.zap()]
LC(10) [A, A.wiff(int)]
LC(11) [B, A.wiff(int)]
LC(18) [B, B.val()]
LC(19) [B, A.val()]

Figure 1. (a) Original and edited version of the 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. (c) Test results for both versions of the example program (here, ✔ indicates that a test is passing, and ✘ indicates that a test is failing). (d) Lookup changes indicating changes to dynamic dispatch.

class A defines (from top to bottom) 5 methods: A.zip(), A.foo(), A.bar(), A.baz(), and A.val(), each returning an integer value. Furthermore, in the original program version, the subclass B of class A contains 3 methods B.waff(), B.bla(), and B.val(). The test suite exercises most of the methods in these classes and compares the values actually returned with the expected values. All the tests in the test suite *pass* on the original program.

In the edited program version, methods A.zap() and A.wiff() are added to class A and several changes are applied to the other methods. In addition, a field A.y is added, the bodies of methods B.waff() and B.bla() are changed, and method B.val() is deleted. A run of the test suite after the edit shows that some of the changes break our assertions, resulting in the *failures* of test1 and test4. Figure 1(c) summarizes the results of running the tests in the original and edited versions of the program.

If, after running the test suite, a developer wants to commit some of these changes to the shared repository without breaking any test, there are two options:

- identify those changes that are not exercised by any test (e.g., the addition of methods or fields that are not yet used), or
- identify changes that are exercised by one or more tests, but that do not contribute to any test failure.

Considering the changes in Figure 1, it is obvious that the method A.wiff() and the field A.y can be added without breaking any tests, because none of the tests exercises this new functionality. However, identifying the remaining changes that can be committed requires a deeper understanding of the program.

For example, the reader may observe that test3 is the only test that exercises the change to the body of method B.waff(), which requires the addition of field A.y. A careful examination reveals that these two changes can be committed safely because test3 passes in both versions of the program and because committing these changes will not affect the behavior of the other tests.

In general, as programs become larger and more complex, the effects of changes on program behavior become harder to understand. Dependences between changes complicate the analysis because a change that is not responsible for any test's failure may still be non-committable because it may be dependent on a responsible change.

3 Change Impact Analysis

This section reviews a change impact analysis [20, 16] that is used as the basis for the *Safe-commit* algorithm that will be presented in Section 4. The change impact analysis used in this paper [20, 19, 21, 24] consists of two steps: (i) a decomposition of the edit into atomic changes, and (ii) correlating these changes with dynamic call graphs that are obtained by running a test suite on the two program versions. Our analysis makes the following common assumptions:

- Tests are deterministic, so that multiple runs produce the same dynamic call graph and the same outcome.
- The execution of a test is independent from the executions of other tests, (e.g., a test must not use data stored in a global object by a previously executed test).
- There are no changes to the environment and libraries between executions of tests.

This section reviews the change model and defines classifications of changes and tests that will be used in Section 4.

3.1 Change Model

A software edit can be decomposed into a set of *atomic changes*. We use a fairly coarse-grained change model that reflects the semantics of an object-oriented program. It supports change categories such as added classes (AC), deleted classes (DC), added methods (AM), deleted methods (DM), changed method bodies (CM), added fields (AF), deleted fields (DF), and lookup changes (LC) (i.e., changes to dynamic dispatch) [20]. Regarding changes to method bodies (CM changes), note that we generate *one* CM change regardless of the number of statements within the respective method's body that have been changed, as we employ a method-level analysis.

In Figure 1, the developer adds method A.zap() to class A as part of the edit that leads to the new version of the program. This method addition is expressed as four atomic changes, including the addition of an empty method, AM(2), and a change to the method's body, CM(3), as shown in the shaded box label. The remaining two atomic changes associated with this method addition, LC(4) and LC(5), represent the effect of the method addition on dynamic dispatch behavior, specifically, the newly possible dispatches of method A.zap() on objects of types A and B, respectively. There are many other kinds of edits that may also impact dispatch behavior. For example, the removal of method B.val() gives rise to the atomic changes LC(18) and LC(19), which correspond to the *changed* dispatch behavior of method A.val() when invoked on objects of type B (originally, such calls dispatched to B.val(), but after the edit they dispatch to A.val()), and the *removed* dispatch behavior of method B.val() on objects of type B, respectively. Figure 1(d) shows all of the LC changes corresponding to the edits shown in Figure 1(a).

3.2 Dependences between changes

An atomic change may be dependent on one or more other atomic changes, that must be applied also in order for the resulting program to compile [19]. Intuitively, an atomic change c_1 *structurally depends* on another atomic change c_2, if applying c_1 to the original version of the program without also applying c_2 results in an invalid program. These dependences can be used to construct syntactically valid intermediate versions of the program that contain some, but not all of the atomic changes, as described in detail in [7, 19].

For example, consider the deleted method B.val() in the example program of Figure 1. This method is referenced in the body of method B.bla(), which has been edited to remove the reference to B.val(). The structural dependence between the corresponding atomic changes DM(16) and CM(15) reflects the fact that DM(16) can only be committed together with CM(15): Committing the deletion of B.val() without committing the change to the body of B.bla() results in a program with a compilation error.

Structural dependences only capture the requirements for creating a syntactically valid program, and do not capture all effects of an edit on program behavior. Certain changes *indirectly* impact program behavior. For example, the addition of a virtual method may give rise to changes in dispatch behavior, and changing a field initializer may result in an implicit change to the bodies of the constructors for the class in which the field is declared. Such effects are captured by *mapping dependences* between changes. Mapping dependences are symmetrical, (i.e., if c_1 is mapping-dependent on c_2, then c_2 is mapping-dependent on c_1) because it is not possible to apply one without the other.

For example, the addition of method A.zap() (AM(2) in Figure 1) causes two LC changes, LC(4) and LC(5), that witness its impact on dispatch behavior. These two LC

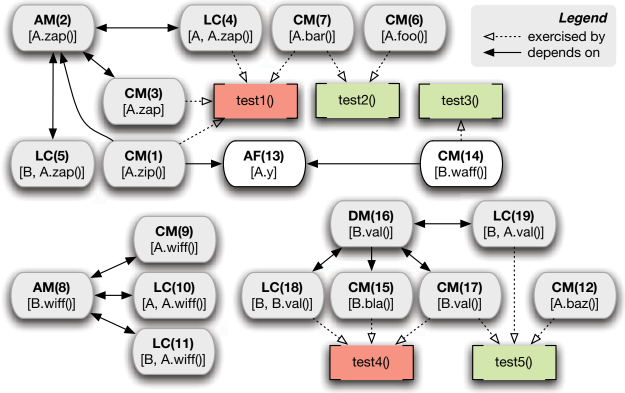


Figure 2. Atomic changes, dependences between atomic changes, and exercised changes for the example program of Figure 1.

changes indicate that this method addition may alter the outcome of tests that exercise this dispatch behavior.

In the remainder of the paper, structural and mapping dependences are not further distinguished, and we will simply write $c_1 \leadsto c_2$ if c_1 is structurally dependent on c_2 or if c_1 is mapping dependent on c_2. Figure 2 shows the atomic changes and dependences for the program of Figure 1.

3.3 Call Graphs and Exercised Changes

Our algorithm for computing committable changes relies on two sets of (dynamic) call graphs. First, the original test suite is executed using the original version of the program and, then the edited version of the test suite is applied to the edited program version. A dynamic call graph is obtained for each test in each version. Figure 3 shows the call graphs for each of the five tests in the original and edited versions. *Bold* boxes and arrows indicate added nodes and edges in the edited version, whereas *dashed* boxes and arrows indicate deletions.

If a test t fails (or crashes[2]), then we determine the subset of atomic changes that may have impacted t's behavior. In particular, we identify a set of *exercised changes* by correlating the computed call graphs with the set of atomic changes: Each CM or LC change that corresponds to a node or edge in t's call graph in either version is exercised by t. We will use *ExercisedChanges*(t) to denote the set of changes exercised by test t. Figure 2 visualizes the exercised changes for each test by way of dashed arrows. In Figure 3, the exercised changes for each test are shown as labels attached to nodes and edges in the call graph. For example, for `test1`, the exercised changes are CM(1), CM(3), LC(4), and CM(7).

3.4 Test Classification

We classify tests based on whether or not they exist in the original and edited versions of the program, and on their outcome in each program version. Each test falls into one of the following categories:

pass→pass pass in both program versions,
fail→pass fail in the original version, and pass in the edited version,
pass→fail pass in the original version, and fail in the edited version,
fail→fail fail in both program versions,
∅→pass added and pass in the edited version,
∅→fail added and fail in the edited version,
pass→∅ deleted and pass in the original version, and
fail→∅ deleted and fail in the original version.

The *pass*→∅ and *fail*→∅ categories are only mentioned for completeness. Deleted tests no longer exercise program behavior, and therefore do not play a role in the computation of committable changes in Section 4.

According to this classification, `test1` and `test4` in the example program are in the *pass→fail* category, and `test2`, `test3` and `test5` are in the *pass→pass* category.

3.5 Change Coverage

Finally, we define a notion of change coverage that will be used for two purposes. First, we will use this notion to conservatively approximate the impact of applying changes on test behavior in the following sense: If two tests t and t' cover non-intersecting sets of changes, then the outcome of t cannot be affected by committing the changes covered by t' and *vice versa*.

Second, we will use the total number of changes covered by *any* test for measuring the effectiveness of our algorithm. Intuitively, our goal is to use a metric that differentiates changes that are "used by tests" from changes that are completely untested (otherwise, a scenario in which a very low percentage of an application's code is covered by tests is likely to give rise to an artificially high percentage of committable changes).

From the discussion of the change model, it is clear that changes cannot be applied or tested in isolation because of their interdependences, and our notion of change coverage takes this into account. Specifically, we will say that a change c is *covered by a test* t if (i) c is exercised by t, (ii) $c \leadsto c'$, and c' is covered by t, or (iii) $c' \leadsto c$, and c' is covered by t. We will write *CoveredChanges*(t) to denote the set of changes covered by test t.

Once all covered changes are computed, we calculate those remaining *uncovered* changes that are not covered by

[2] *JUnit* distinguishes between assertion failures and exceptions; both are treated as test failures by our algorithm.

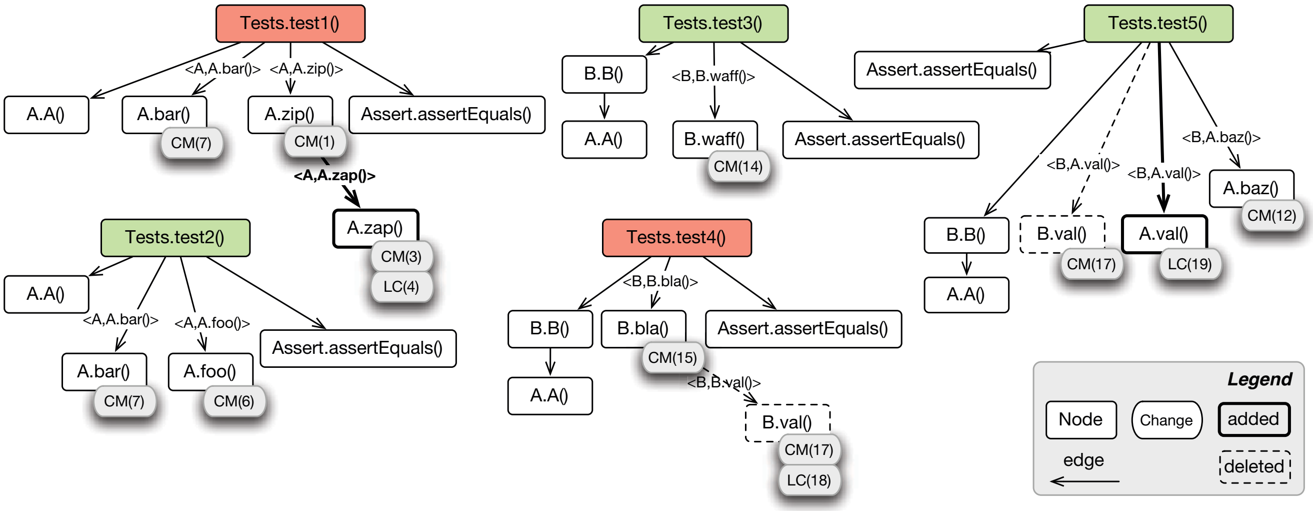


Figure 3. Call graphs for the tests in the original and edited versions of the example program. *Bold* **boxes and arrows indicate added nodes and edges in the edited version,** *dashed* **boxes and arrows indicate deletions. Gray annotations to the boxes refer to the atomic changes shown in Figure 1.**

any test. In the example program, the newly added method `A.wiff()` is not exercised by the test suite, thus AM(8), CM(9), LC(10) and LC(11) are uncovered changes.

4 Determining Committable Changes

Figure 4 shows our algorithm for computing a set of committable changes. The algorithm takes as inputs: (i) the set *AC* containing all atomic changes, with the dependence relation $\leadsto$ that was defined in Section 3, (ii) a set *InputTests* of tests that exhibit unwanted behavior in the edited version of the program (according to the selected commit policy), and for each test t, (iii) the set $\mathit{CoveredChanges}(t)$ of changes covered by that test, and (iv) its set of exercised changes $\mathit{ExercisedChanges}(t)$. The output of the algorithm is a set *CommitChanges* of changes that can be committed safely.

The algorithm follows an iterative workset-based approach to simultaneously identify sets of *non-committable changes* and *preserved tests* for which the original behavior must be preserved. For each non-committable change, the set *NewNonCommitChanges* contains non-committable changes that are identified in the current iteration, and the set *ProcessedNonCommitChanges* contains non-committable changes that have already been processed (i.e., their impact on test behavior has been explored fully). Likewise, there are sets *NewPreservedTests* and *ProcessedPreservedTests* of preserved tests that are identified in the current iteration, and preserved tests that have already been processed.

An important property of the algorithm is that for each test t, either all changes that impact t's behavior are committable (along with the changes that depend on them via the $\leadsto$ relation), or all such changes are non-committable. Intuitively, this means that such a test t will have either the behavior that it had in the original version program, or the behavior that it has in the edited version.

The algorithm begins on lines 1–4 by initializing *NewPreservedTests* to *InputTests*, and initializing each of *ProcessedPreservedTests*, *NewNonCommitChanges*, and *ProcessedNonCommitChanges* to the empty set.

The algorithm then traverses the outer while-loop (lines 5–31) as long as new tests are found for which the original behavior must be preserved. Each such test t is removed from *NewPreservedTests* and added to *ProcessedPreservedTests* (lines 8–9), and each LC or CM change c that is exercised by test t, and that is not already marked as non-committable is added to *NewNonCommitChanges* (lines 10–14).

Next, on lines 16–30 the impact of each new non-committable change c on tests is explored. Specifically, each test t that has not already been marked as a preserved test and for which $c \in \mathit{CoveredChanges}(t)$ is added to *NewPreservedTests*.

The purpose of the loop on lines 25–29 is to ensure that applying the identified set of committable changes results in a syntactically valid program with correct test behavior. To this end, the algorithm marks as non-committable any atomic change c' that is dependent on a non-committable change c, according to the $\leadsto$ relation defined in Section 3.

Finally, the algorithm computes the set of committable changes as any covered atomic change that is not found to be non-committable (line 32).

```
Input: $AC$: set containing all atomic changes, with dependence
        relation $\rightsquigarrow$
Input: $InputTests$: set of tests with undesirable behavior
Input: $CoveredChanges(t)$: set of changes covered by test $t$
Input: $ExercisedChanges(t)$: the changes exercised by test $t$
Output: $CommitChanges$: set of committable changes

 1 $NewPreservedTests = InputTests$;
 2 $ProcessedPreservedTests = \emptyset$;
 3 $NewNonCommitChanges = \emptyset$;
 4 $ProcessedNonCommitChanges = \emptyset$;
 5 while $NewPreservedTests \neq \emptyset$ do
 6     while $NewPreservedTests \neq \emptyset$ do
 7         select a test $t \in NewPreservedTests$;
 8         $NewPreservedTests = NewPreservedTests \setminus \{ t \}$ ;
 9         $ProcessedPreservedTests =$
                   $ProcessedPreservedTests \cup \{ t \}$;
10         foreach change $c$ in $ExercisedChanges(t)$ do
11             if change $c \notin (NewNonCommitChanges \cup$
                       $ProcessedNonCommitChanges)$ then
12                 $NewNonCommitChanges =$
                           $NewNonCommitChanges \cup \{ c \}$;
13             end
14         end
15     end
16     while $NewNonCommitChanges \neq \emptyset$ do
17         select a change $c \in NewNonCommitChanges$;
18         $NewNonCommitChanges =$
                   $NewNonCommitChanges \setminus \{ c \}$;
19         $ProcessedNonCommitChanges =$
                   $ProcessedNonCommitChanges \cup \{ c \}$;
20         foreach test $t$ such that $CoveredChanges(t) \neq \emptyset$ do
21             if $c \in CoveredChanges(t) \wedge (t \notin$
                     $(NewPreservedTests \cup ProcessedPreservedTests))$
               then
22                 $NewPreservedTests =$
                           $NewPreservedTests \cup \{ t \}$;
23             end
24         end
25         foreach change $c' \in AC$ do
26             if $c' \rightsquigarrow c \wedge (c' \notin (NewNonCommitChanges \cup$
                       $ProcessedNonCommitChanges))$ then
27                 $NewNonCommitChanges =$
                           $NewNonCommitChanges \cup \{ c' \}$;
28             end
29         end
30     end
31 end
32 $CommitChanges = AC \setminus$
       $(ProcessedNonCommitChanges \cup UncoveredChanges)$;
```

Figure 4. Algorithm for computing committable changes.

4.1 Modeling Commit Policies

The algorithm of Figure 4 can be used to implement the following three commit policies:

Restrictive. Allow programmers to commit a set of changes if each change in this set is covered only by passing (*pass*→*pass*, *fail*→*pass*, or ∅→*pass*) tests.

Moderate. Allow programmers to commit a set of changes if each change in this set is covered only by *pass*→*pass*, *fail*→*pass*, *fail*→*fail*, ∅→*pass*, or ∅→*fail* tests.

Permissive. Allow programmers to commit a set of changes if each change in this set is uncovered, or if it is covered only by *pass*→*pass*, *fail*→*pass*, *fail*→*fail*, ∅→*pass*, or ∅→*fail* tests.

Informally, the *Restrictive* commit policy allows programmers to commit only those changes that are "successfully tested", i.e., exercised by tests that pass in the edited version of the program. This policy could be used in the later stages of development, when avoiding the introduction of new and untested functionality is crucially important. The *Moderate* commit policy enables programmers to commit changes provided that they are covered and do not break existing tests. Introducing new failing tests is permitted under the *Moderate* policy, in order to enable collaborative development scenarios in which one team member writes and commits tests (that initially fail), and the functionality needed to make the tests pass is implemented later by other team members. The *Permissive* policy extends the *Moderate* policy by allowing programmers also to commit uncovered changes. Both the *Moderate* and *Permissive* commit policies could be used during development instead of the commonly used *conservative* policy without compromising the integrity of the repository.

The algorithm of Figure 4 can be used to compute a set of committable changes that is compatible with the *Restrictive* policy by initializing the set *InputTests* to include all *pass*→*fail*, *fail*→*fail*, and ∅→*fail* tests. By contrast, the *Moderate* commit policy can be implemented by initializing *InputTests* to include all *pass*→*fail* tests. Alternatively, the *Permissive* policy is obtained by initializing *InputTests* to include all *pass*→*fail* tests and by adding the set *UncoveredChanges* to the computed set of *CommitChanges*.

4.2 Example

We will now discuss how the algorithm of Figure 4 is applied to the example of Figure 1(a). Assuming that the *Restrictive* policy is chosen, *InputTests* is initialized to the set { `test1`,`test4` }.

The while-loop on lines 6–15 is executed twice. During the first iteration, t is bound to `test1`, and the execution of the foreach-loop on lines 10–14 results in adding the changes CM(1), CM(3), LC(4), and CM(7) to *NewNonCommitChanges* (these changes are correlated with nodes in the call graph for `test1` that was shown in Figure 3). During the second iteration, t is bound to `test4`, and the execution of the foreach-loop on lines 10–14 results in adding the changes CM(15), CM(17), and LC(18) to

NewNonCommitChanges. When execution reaches line 16, *NewNonCommitChanges* contains the elements CM(1), CM(3), LC(4), CM(7), CM(15), CM(17), and LC(18), *ProcessedNonCommitChanges* is empty, *NewPreservedTests* is empty, and *ProcessedPreservedTests* contains `test1` and `test4`.

Next, the loop on lines 16–30 is entered. Line 17 is executed for each change c in *NewNonCommitChanges*. This results in adding `test2` (because CM(7) $\in$ *CoveredChanges*(`test2`)) and `test5` (because CM(17) $\in$ *CoveredChanges*(`test5`)) to *NewPreservedTests*.

The execution of the loop on lines 25–29 results in adding any change c' to *NewNonCommitChanges* such that $C' \rightsquigarrow C$, for some non-committable change c. By examining the dependences in Figure 2, we determine that AM(2), LC(5), DM(16), and LC(19) are added to *NewNonCommitChanges*. Note that AF(13) is *not* added to *NewNonCommitChanges* because it has no dependence to any non-committable change. Hence, at the end of the first iteration of the outer while loop, we have that *NewNonCommitChanges* contains AM(2), LC(5), DM(16), and LC(19), *ProcessedNonCommitChanges* contains CM(1), CM(3), LC(4), CM(7), CM(15), CM(17), and LC(18), *NewPreservedTests* contains `test2` and `test5`, and *ProcessedPreservedTests* contains `test1` and `test4`.

In the second iteration of the outer while loop, CM(6) is added to *NewNonCommitChanges* because CM(6) $\in$ *CoveredChanges*(`test2`) and CM(12) is added to *NewNonCommitChanges* because CM(12) $\in$ *CoveredChanges*(`test5`). During the execution of the loop on lines 16–30, no additional preserved tests are identified. No additional changes are found during the execution of the loop on lines 25–29. So at the end of the second iteration of the outer while loop, *NewNonCommitChanges* and *NewPreservedTests* are empty, *ProcessedPreservedTests* contains `test1`, `test2`, `test4`, and `test5`, and *ProcessedNonCommitChanges* contains CM(1), AM(2), CM(3), LC(4), LC(5), CM(6), CM(7), CM(12), CM(15), DM(16), CM(17), LC(18) and LC(19).

Finally, the set of committable changes is computed on line 32 as AF(13) and CM(14). Intuitively, these changes are committable under the *Restrictive* policy because they are successfully tested by `test3`.

5 Implementation and Evaluation

5.1 Implementation

JUnitMX is built as an extension of the widely used *JUnit* plug-in for Eclipse, enabling developers already familiar with *JUnit* and the *Eclipse Java Development Tools*[3] to leverage their experience with these tools. JUnitMX requires two versions of the program: the *original version*, for which the CVS HEAD version is used by default, and the *edited version*, for which the current version in the developer's local workspace is used. In order to use the tool, developers must select a standard *JUnit* test suite in the edited version and run it using a special launch configuration; note that JUnitMX runs the tests associated with both versions. JUnitMX enables developers to compute committable changes according to the *Restrictive*, *Moderate*, and *Permissive* policies that were presented earlier. In addition, all standard *JUnit* functionality is still available.

JUnitMX hooks into the execution of a *JUnit* test suite and adds a pre- and a post-processing phase. In the pre-processing phase, JUnitMX uses Chianti[4] [20, 19], a tool that was previously developed by our group. Chianti creates an abstract syntax tree (AST) for the classes in each version and compares their structure to obtain the set of atomic changes to construct the change model as presented in Section 3. In addition, JUnitMX uses Dila, a library for efficiently constructing dynamic program representations such as call graphs, that we developed specifically for this project[5]. Dila uses a custom class loader to instrument the target application's classes before they are executed. The purpose of the added instrumentation code is to construct a separate dynamic call graph on-the-fly, during the execution of each test. In its post-processing phase, JUnitMX performs the analyses for computing committable changes that was presented in Section 4. An intermediate version of the program can be constructed from the computed set of committable changes by adding only those changes to the original program version. A subsequent run of the test suite on this version can validate the correctness of our committable change analysis.

5.2 Goals and Experimental Setup

In cases where all tests pass, every change that is covered by the tests is committable. Our evaluation therefore focuses on version pairs with worsening tests that fail in the edited version. The major goal of the evaluation is to show that the percentage of committable changes is significant even in the presence of test failures, so that development teams have a significant benefit from adopting the presented approach.

As mentioned in previous papers [19, 7], failing tests are rarely found in the versions that are checked into public repositories. We analyzed 6 version pairs with failing tests of *Daikon*, a dynamic invariant detector developed by M. Ernst [9]. Each version pair contains a week's worth of changes during the year 2002.

[3] http://www.eclipse.org/jdt/

[4] http://www.prolangs.rutgers.edu/projects/chianti/

[5] http://www.prolangs.rutgers.edu/projects/dila/

Project	**Daikon**						
Version Pair	**Average**	**p1** 01/07-01/14, 2002	**p2** 01/14-01/21, 2002	**p3** 01/21-01/28, 2002	**p4** 01/28-02/04, 2002	**p5** 04/15-05/06, 2002	**p6** 11/11-11/19, 2002
Tests (original)		40	42	42	42	52	62
Tests (edited)		42	42	42	42	60	59
Tests (pass-->pass)		36	37	37	37	52	57
Tests (pass-->fail)		0	1	0	0	0	2
Tests (fail-->pass)		0	0	0	0	0	0
Tests (fail-->fail)		4	4	5	5	0	0
Tests (new-->pass)		2	0	0	0	0	0
Tests (new-->fail)		0	0	0	0	8	0
Changes (total)		1751	274	1485	614	302	6050
Changes (covered)		1013	5	1225	20	130	185
Restrictive Policy							
Tests (selected)		4	5	5	5	8	2
Changes (committable)		111	4	26	6	50	28
Committable (% of total)	**4.6%**	6.3%	1.5%	1.8%	1.0%	16.6%	0.5%
Committable (% of covered)	**29.4%**	11.0%	80.0%	2.1%	30.0%	38.5%	15.1%
Moderate Policy							
Tests (selected)		0	1	0	0	0	2
Changes (committable)		1013	4	1225	20	130	28
Committable (% of total)	**31.4%**	57.9%	1.5%	82.5%	3.3%	43.0%	0.5%
Committable (% of covered)	**82.5%**	100.0%	80.0%	100.0%	100.0%	100.0%	15.1%
Permissive Policy							
Tests (selected)		0	1	0	0	0	2
Changes (committable)		1751	273	1485	614	302	5893
Committable (% of total)	**99.5%**	100.0%	99.6%	100.0%	100.0%	100.0%	97.4%

Table 1. Data gathered for the selected version pairs of Daikon.

5.3 Experimental Results

Table 1 summarizes the results obtained for *Daikon*. The columns of the table correspond to version pairs of *Daikon* (identified by their dates), with one additional column for each application that provides averages where appropriate. The rows of the table show, from top to bottom: the number of tests in the original version, the number of tests in the edited version, and a further breakdown of the tests by outcome into the categories we discussed previously (*pass→pass*, *pass→fail*, etc.). Next, the table shows 3 sets of rows for the results of our analysis, one for each of the *Restrictive*, *Moderate*, and *Permissive* commit policies. For each policy, we show the number of tests selected as input to the algorithm of Figure 4, the total number of committable changes, the number of committable changes as a percentage of all changes, and the number of committable changes as a percentage of covered changes (the latter is omitted for the *Permissive* policy where this metric does not make sense, because uncovered changes also are committed).

We found that under the *Restrictive* policy, an average 4.6% of all atomic changes could be committed. For the *Moderate* policy, the average percentage of committable changes among all changes is 31.4%. Finally, for the *Permissive* policy, we found an average of 99.5% of all changes to be committable.

From Table 1, it is clear that often the majority of changes is not covered. In projects with better test coverage, higher percentages of committable changes can be expected for the *Restrictive* and *Moderate* policies. Therefore we also report the number of committable changes as a percentage of *covered* changes because it provides a better measure of the accuracy of our analysis. The *Restrictive* policy identifies 29.4% and the *Moderate* policy 82.5% of covered changes as committable. Even though further experimentation is needed to validate the usefulness of each commit policy, we have shown evidence that a significant percentage of changes can be released safely to a repository. In the deployment scenario we envision, programmers would apply the analysis frequently—perhaps as part of each test run—resulting in significantly smaller differences between successive versions. We conjecture that the percentage of committable changes would increase in this scenario.

On average, our analysis added a total overhead of 4 minutes to the execution of the tests in both versions. The actual overhead for each version pair varied from 1 to 15 minutes. It is dominated by the computation of covered changes for each test which becomes more expensive as the size of the call graphs or the number of changes increases[6]. We are currently working on optimizations that are likely to reduce this overhead considerably.

5.4 Experiments with Releases

We also analyzed 5 public releases of *JMeter*[7], taken from the SIR repository [8]. These releases are separated

[6] All performance data were measured on an Apple MBP Laptop Computer with 2.6 GHz Intel Core 2 Duo processor and 2GB main memory.

[7] http://jakarta.apache.org/jmeter/

by time intervals of up to 4 months and by huge numbers of atomic changes (up to 17475). As another indication of the amount of change, we found that, on average, 31.4% of the methods was changed between releases. In other words, this is a scenario for which we did not envision our technique to be applicable. For each release, SIR provides a number of artificial faults that—when seeded into that version—result in a number of test failures. In our experiments, we used 4 version pairs to determine the changes that can be committed when comparing the unmodified version N of *JMeter* with version $N+1$ into which all provided faults have been seeded.

To our surprise, the *Restrictive* and *Moderate* still managed to identify an average of 3.8% and 4.0% of the covered changes as committable, despite the huge amount of change between releases. The *Permissive* policy even identified 46.0% of all changes as committable, because a significant percentage of the changes was uncovered.

6 Related Work

Revision Control, Software Merging. Traditional *pessimistic* revision control system such as SCCS [12] and RCS [25] prevent *direct merge conflicts* by allowing only one developer to have a writable copy of any artifact at any given time. Most modern revision control systems are *optimistic* in the sense that they allow multiple developers to modify a file concurrently. For example, CVS [4] and Subversion [18] allow developers to modify artifacts concurrently. Whenever a developer wants to commit changes, the local copy must be reconciled with the current "head" of the repository. Conflicting changes are merged by a simple textual merging algorithm if they do not involve overlapping regions of files, and the user must manually resolve conflicts otherwise. The only conflicts detected by CVS and SCCS are syntactic ones in which two developers are editing the same textual region of a file. However, *indirect merge conflicts*, where a developer's changes to one file adversely affect the changes by another developer to another file may still arise. In our work, the preservation of behavior of the tests in a test suite is used as an oracle to establish the absence of indirect merge conflicts.

We are not aware of any previous work where program analysis is used to determine subsets of changes that can be committed safely. However, there has been a significant amount of research on problems related to the merging of software artifacts [15]. Some of this work involves the use of program analysis to determine whether or not a given set of changes can be integrated into a program *in its entirety* without affecting behavior. In work by Binkley et al. [5], static slicing [13] serves as the basis for an algorithm that integrates changes in variants of a program in a way that is guaranteed to preserve behavior. In cases where preservation of behavior cannot be guaranteed, their technique simply reports that interference was detected.

Workspace Awareness Tools. The goal of workspace awareness tools is to make developers aware of each other's changes before these are committed to a central repository, so that they can take proactive steps to prevent or minimize unforeseen interferences and/or duplicative development. Such steps may include talking to other developers, reassigning tasks, and postponing changes until the other developer has done a commit.

Palantír [23] is a workspace awareness tool that increases awareness by continuously sharing, among a team of developers, information about changes and an estimate of their severity via a graphical user-interface. This information is captured and shared at the level of events such as POPULATED (indicating that an artifact has been placed in a developer's workspace), CHANGESCOMMITTED (a new version of an artifact has been stored in the repository), and SEVERITYCHANGED (the amount of change—e.g., as the percentage of lines of code changed—has changed significantly). Palantír initially only supported functionality for reporting *direct conflicts* (i.e., situations in which two or more developers edit the same artifact) [23], but was recently extended with support for a limited class of *indirect conflicts*. In this work, syntactic information about dependences between artifacts is used to determine if changes to different artifacts may interfere with each other [22].

Cheng et al. [6] discuss a "Concert Awareness" feature of the Jazz environment that visualizes what other developers are doing with their local copies of files. Estublier and Garcia [10] point out the importance of considering semantic dependences between artifacts in different files in the context of workspace awareness tools. Their tool, Celine, also takes into account factors such as the workspace topology and the cooperative engineering policy that is being used. The Hipikat system [26] aims to reduce parallel development by providing a facility that recommends artifacts related to a specific task. Hipikat's generates recommendations by performing a textual similarity analysis of CVS repositories, issue-tracking systems (e.g., Bugzilla), newsgroups, and web sites associated with the project.

Our techniques complement workspace awareness tools by helping developers prevent the premature release of changes that may hamper others.

Continuous Integration. To avoid merge conflicts, it has long been known that it is advisable to "commit early, and commit often" [3]. Development methodologies such as *continuous integration* [11] advocate that team members commit their work frequently using an automated build process that includes running tests. Team members who experience test failures in their local workspace are discouraged from committing their changes. Our research is well-aligned with continuous integration because it enables programmers to commit (some of) their changes earlier, even when there are failing tests in their local workspace.

7 Conclusions

In current practice, developers often postpone the release of their changes until all tests pass in their local workspace in order to preserve code quality. This increases the time intervals between commits, thereby increasing the risk of merge conflicts and duplicative development later on.

We have presented an analysis-based technique for determining changes that can be committed without compromising the code base in the repository, even in cases where there are failing tests in a developer's local workspace. Our algorithm, *Safe-commit*, is based on a previously developed change impact analysis [20, 19, 21, 24], and we show how it can be used to implement 3 new commit policies (*Restrictive*, *Moderate*, and *Permissive*) with varying levels of strictness. This enables developers to release their changes more quickly, thus reducing the risk of duplicative development and merge conflicts.

We measured the effectiveness of the new commit policies using versions of *Daikon* with associated failing tests. In this experiment, an average of 4.6%, 31.4%, and 99.5% of all changes were identified as committable according to the *Restrictive*, *Moderate*, and *Permissive* commit policies, respectively. In another experiment, we applied *Safe-commit* to public releases of *JMeter*. To our surprise, a nontrivial number of committable changes was identified, despite a huge number of changes that separates these releases.

References

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

[2] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, 2004.

[3] D. Berlin and G. Rooney. *Practical Subversion, Second Edition*. Apress, 2006.

[4] B. Berliner. CVS II: Parallelizing software development. In *Proc. USENIX Winter 1990 Technical Conf.*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[5] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, 1995.

[6] L.-T. Cheng, S. Hupfer, S. Ross, and J. Patterson. Jazzing up eclipse with collaborative tools. In *Proc. 2003 OOPSLA workshop on eclipse technology eXchange*, pages 45–49, 2003.

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

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

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

[10] J. Estublier and S. Garcia. Process model and awareness in scm. In *Proc. 12th International Workshop on Software Configuration Management (SCM'05)*, pages 59–74, 2005.

[11] M. Fowler. Continuous Integration. `www.martinfowler.com/articles/continuousIntegration.html`, 2006.

[12] A. L. Glasser. The evolution of a source code control system. In *Proc. software quality assurance workshop on functional and performance issues*, pages 122–125, 1978.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12(1):26–61, 1990.

[14] KDE Development Team. Commit Policy of the KDE Project. `techbase.kde.org/Policies/SVN_Commit_Policy`.

[15] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. on Softw. Eng.*, 28(5):449–462, May 2002.

[16] A. Orso, T. Apiwattanapong, and M. J. Harrold. 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, Sept. 2003.

[17] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, 2001.

[18] M. Pilato. *Version Control With Subversion*. O'Reilly & Associates, Inc., 2004.

[19] X. Ren, O. C. Chesley, and B. G. Ryder. Identifying failure causes in Java programs: An application of change impact analysis. In *IEEE Trans. on Softw. Eng.*, volume 32, pages 718–732, 2006.

[20] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for practical change impact analysis of Java programs. In *Proc. Conf. on Object Oriented Programming, Systems and Applications (OOPSLA'04)*, pages pp 432–448, Oct. 2004.

[21] B. G. Ryder and F. Tip. Change Impact Analysis for Object-oriented Programs. In *Proc. Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 46–53, 2001.

[22] A. Sarma, G. Bortis, and A. van der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proc. 22nd IEEE/ACM International Conf. on Automated Software Engineering (ASE'07)*, pages 94–103, 2007.

[23] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: raising awareness among configuration management workspaces. In *Proc. 25th International Conf. on Software Engineering (ICSE'03)*, pages 444–454, Washington, DC, USA, 2003.

[24] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding Failure-inducing Changes in Java Programs Using Change Classification. In *Proc. 14th Symp. on the Foundations of Software Engineering (FSE-14)*, pages 57–68, Portland, OR, USA, Nov. 7–9, 2006.

[25] W. F. Tichy. RCS — a system for version control. *Software — Practice and Experience*, 15(7):637–654, 1985.

[26] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proc. 25th International Conf. on Softw. Engineering (ICSE'03)*, pages 408–418, 2003.