Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis

Xiaoxia Ren, Ophelia C. Chesley, and Barbara G. Ryder, *Member*, *IEEE Computer Society*

Abstract—During program maintenance, a programmer may make changes that enhance program functionality or fix bugs in code. Then, the programmer usually will run unit/regression tests to prevent invalidation of previously tested functionality. If a test fails unexpectedly, the programmer needs to explore the edit to find the failure-inducing changes for that test. *Crisp* uses results from *Chianti*, a tool that performs semantic change impact analysis [1], to allow the programmer to examine those parts of the edit that affect the failing test. *Crisp* then builds a compilable intermediate version of the program by adding a programmer-selected partial edit to the original code, augmenting the selection as necessary to ensure compilation. The programmer can reexecute the test on the intermediate version in order to locate the exact reasons for the failure by concentrating on the specific changes that were applied. In nine initial case studies on pairs of versions from two real Java programs, *Daikon* [2] and *Eclipse jdt compiler* [3], we were able to use *Crisp* to identify the failure-inducing changes for all but 1 of 68 failing tests. On average, 33 changes were found to affect each failing test (of the 67), but only 1-4 of these changes were found to be actually failure-inducing.

Index Terms—Fault localization, semantic change impact analysis, edit change dependence, regression testing, intermediate versions of programs.

1 Introduction

Regression tests are developed by programmers over time to confirm the fundamental functionalities of a program after it has been changed. After a long code editing session, regression tests are executed to ensure that the updated program version works properly with respect to previous releases. During this phase, any test that produces unexpected results may indicate potential defects introduced by the edit that created the updated version. When a test fails, a programmer is burdened with the task of searching through the program for the source(s) of the failure. Moreover, a failure can be caused by nontrivial combinations of changes.

In this paper, we describe *Crisp*, a tool to assist programmers in isolating relevant portions of an edit that directly cause the failure of a regression test. This work leverages and augments our earlier research prototype *Chianti*, an Eclipse plug-in that performs semantic change impact analysis of Java programs [4], [1]. *Chianti* divides a program edit into its constituent parts (known as *atomic changes*), identifies a set of tests that are impacted by these changes and, for each *affected test*, identifies the subset of changes (called *affecting changes*) that affect its behavior. In essence, *Chianti* automatically identifies all the relevant changes pertaining to each regression test. Initial experiments with *Chianti* have evidenced promising results for these analyses [1].

While the set of affecting changes of an affected test can be small relative to the total number of atomic changes, examining each of these changes and pinpointing the few that induce the failure of a test is a tedious task if performed manually. For large applications, the parts of an edit are interrelated in many ways, and there can be more than one subset of changes that the programmer considers as failure-prone with respect to a specific test.

There are benefits in guiding the iterative process of selecting changes of interest and applying them to the original software version to create intermediate program versions. Each of these versions can then be tested using the tests that failed earlier. Programmers can ignore those changes that do not result in failure and then further examine and isolate smaller sets of changes until they locate those that directly cause the failure. Our goal is to provide programmers with a tool to aid in this process, in which the programmer does not need to be concerned with the syntactic interrelationships of the changes nor with manually editing any code.

Our previous papers presented the overall theoretical semantic change analysis framework [4] and our empirical experiences with the *Chianti* prototype [1] built as an *Eclipse* plug-in. The example in Section 2 illustrates the main ideas of the semantic change analysis framework presented in these previous papers. Our initial description of *Crisp* [5], an *Eclipse* plug-in built to work with *Chianti*, discussed how to use the change dependences reported by *Chianti* to form intermediate program versions. It also reported the small Daikon case study that is included here.

This paper substantially extends the work previously presented. The change dependences presented in this paper are both a refinement and an augmentation of those previously discussed; we have categorized them into three kinds reflecting their usage in *Crisp*. In addition to the first case study, we include a larger case study of *Eclipse jdt compiler* in which, by using *Crisp*, we were able to identify failure-inducing changes in 65 of the 66 tests.

In summary, this paper makes the following additional contributions over our previous conference publications:

• *The authors are with the Division of Computer and Information Sciences, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019. E-mail: {xren, ochesley, ryder}@cs.rutgers.edu.*

Manuscript received 3 Jan. 2006; revised 28 Mar. 2006; accepted 7 Apr. 2006; published online 27 Sept. 2006.
Recommended for acceptance by T. Gyimothy and V. Rajlich.
For information on obtaining reprints of this article, please send e-mail to tse@computer.org, and reference IEEECS Log Number TSE-0002-0106.

- definitions of refined dependence relationships among the changes in an edit, by exploring a richer set of dependences than was originally considered, and by discussing the limitations of the current framework that made *Crisp* fail to build a valid intermediate program version automatically once during our experiments, and
- a new, larger case study using *Eclipse jdt compiler* for which *Crisp* successfully built the valid intermediate versions and helped to pinpoint the failure-inducing changes in 65 of 66 failing tests.

The remainder of this paper is organized as follows: Section 2 introduces a running example that illustrates the change impact analysis process, which finds the affecting changes set for each affected test and the subsequent exploration using *Crisp* to identify the failure-inducing changes. In Section 3, we summarize the underlying semantic change impact analysis and discuss the kinds of dependences between atomic changes. Section 4 reports on some engineering issues encountered in building our tool. Section 5 shows how we have applied *Crisp* in two case studies. Section 6 discusses the limitations of our current framework. Finally, related work and conclusions are summarized in Sections 7 and 8, respectively.

2 Locating Failure-Inducing Changes Using *Chianti* and *Crisp*

We will use the example program of Fig. 1 to illustrate our approach. Fig. 1a shows two versions of the program. 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. Each box is labeled with the numbers of the corresponding atomic changes.

Associated with the program are three JUnit tests [6], `Tests.test1`, `Tests.test2`, and `Tests.test3`, which are shown in Fig. 1b. Note that it is assumed that these tests will be used with both versions of the program.

2.1 Change Impact Analysis

Our change impact analysis[1] relies on the computation of a set of atomic changes that capture all source code modifications at a semantic level that is amenable to practical analysis. We currently use a fairly coarse-grained model of atomic changes, where changes are categorized as added classes (**AC**), deleted classes (**DC**), added methods (**AM**), deleted methods (**DM**), changed methods (**CM**), added fields (**AF**), deleted fields (**DF**), and lookup (i.e., dynamic dispatch) changes (**LC**). (There are a few more categories of atomic changes that are not relevant for the example under consideration that will be discussed in Section 3.)

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

Fig. 1c shows the atomic changes corresponding to 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 and the bottom half shows the method or field involved. An arrow from an atomic change A_1 to an atomic change A_2 indicates that A_1 is dependent on A_2.[2] Consider, for example, the addition of the call to method `inc()` in method `B.foo()` (i.e., **CM** atomic change 4 in Fig. 1c). Observe that changing this method would lead to a syntactically invalid program unless method `B.inc()` is also added (i.e., **AM** atomic change 1). Therefore, atomic change 4 is dependent on atomic change 1. The observant reader may have noticed that there is also a **CM** change for method `B.inc()` (i.e., **CM** atomic change 2). This is the case because our method for deriving atomic changes decomposes the source code change of adding method `B.inc()` into two steps: the addition of an empty method `B.inc()` (i.e., **AM** atomic change 1), and the insertion of the body of method `B.inc()` (i.e., **CM** atomic change 2), where the latter is dependent on the former. Notice that our model of dependences between atomic changes correctly captures the fact that adding the call to `B.foo()` requires adding the declaration of method `B.inc()`, but does not require adding the field `A.x`.

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 runtime type `Y` results in the selection of a different method after the edit. Consider, for example, the addition of method `C.bar()` (i.e., **AM** atomic change 9) to the original program. As a result of this change, a call to `A.bar()` on an object of type `C` will dispatch to `C.bar()` in the edited program, whereas it used to dispatch to `A.bar()` in the original program. This change in dispatch behavior is captured by **LC** atomic change 11.

Fig. 1d shows the dynamic call graphs[3] for the three tests `test1`, `test2`, and `test3`, before the changes have been applied. In these call graphs, edges corresponding to dynamic dispatch are labeled with a pair $\langle \texttt{T}, \texttt{M} \rangle$, where `T` is the runtime type of the receiver object and `M` is the method shown as invoked at the call site. A test is determined to be affected if its call graph in the original version of the program contains a node that corresponds to a changed method **CM** or deleted method **DM** or contains an edge that corresponds to a lookup change **LC**. Using the call graphs in Fig. 1d, it is easy to see that 1) `test1` is not affected and 2) `test2` and `test3` are affected because their call graphs each contain a node for `B.foo()`, which corresponds to **CM** atomic change 4.

Call graphs for the affected tests on the edited version of the program are shown in Fig. 1e. Only call graphs for `test2` and `test3` are needed, since `test1` is not affected by any of the changes. The set of atomic changes that affect a given affected test includes 1) all atomic changes for changed methods (**CM**) that correspond to a node in the call graph (in the edited program), 2) atomic changes in the lookup change (**LC**) category that correspond to an edge in the call graph (in the edited program), and 3) their transitively prerequisite atomic changes.

We can compute the affecting changes for `test3` as follows: Observe, that the call graph for `test3` in Fig. 1e contains methods `B.foo()`, and `C.bar()`, and an edge labeled $\langle \texttt{C}, \texttt{A.bar()} \rangle$. Node `B.foo()` corresponds to atomic

1. Here we present intuitive definitions of our change impact analysis that pertain to *Crisp* usage; a more formal treatment is available in [1].

2. The different line styles represent kinds of dependences that will be discussed in Section 3.

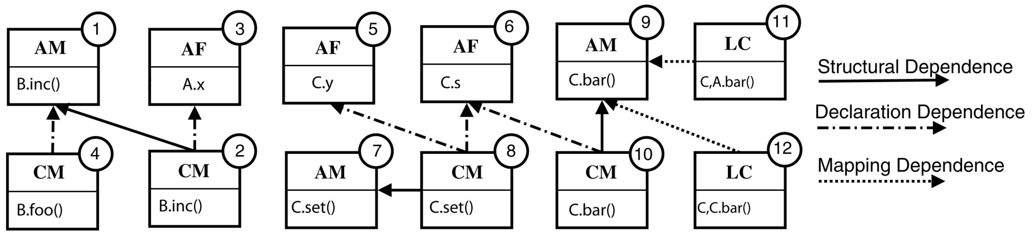
3. *Chianti* can work with call graphs that have been constructed using static analysis or from the actual execution of the tests.

(a)

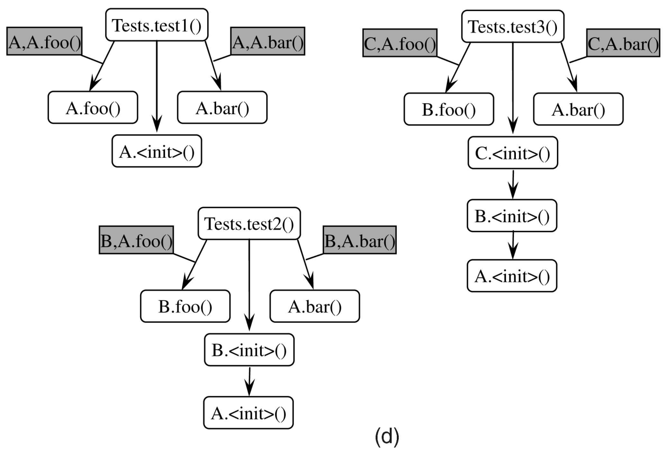
```
class A    {
   public int i, s,        x^3;
   public void foo()       { }
   public void bar()       {s=i+1;   }
}
class B extends A      {
   public void foo()      {
      if (false) inc();^4
      s = i;
   }
   private void inc()       {x++; }^1,2
}
class C extends B      {
   int y, s;^5,6
   public void bar()      {s++; }^9,10,11,12
   private void set()      {y=s; }^7,8
}
```

(b)

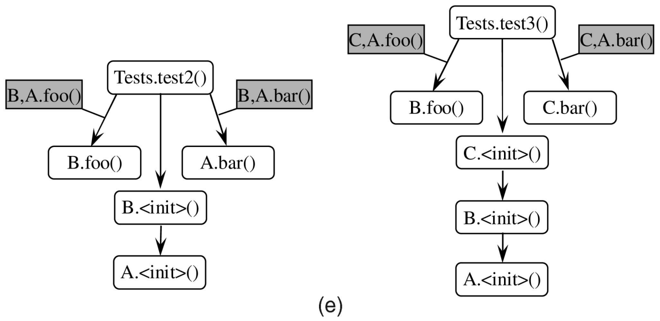
```
public class Tests extends TestCase {
   public void test1()       {
      A a = new A();
      a.foo(); a.bar();
      Assert.assertTrue(a.s > a.i);
   }
   public void test2()       {
      A a = new B();
      a.foo(); a.bar();
      Assert.assertTrue(a.s > a.i);
   }
   public void test3()       {
      A a = new C();
      a.foo(); a.bar();
      Assert.assertTrue(a.s > a.i);
   }
}
```



(c)



(d)



(e)

(f)

```
class A    {
   public int i, s;
   public void foo()       { }
   public void bar()       {s=i+1;   }
   public int x^3;
}
class B extends A      {
   public void foo()      {
      if (false) inc();^4
      s = i;
   }
   private void inc()       {x++; }^1,2
}
class C extends B      { }
```

(g)

```
class A    {
   public int i, s;
   public void foo()       { }
   public void bar()       {s=i+1;   }
}
class B extends A      {
   public void foo()       {s=i;   }
}
class C extends B      {
   int s;^6
   public void bar()      {s++; }^9,10,11,12
}
```

Fig. 1. (a) Original and edited version of example program. (b) Tests associated with the example of (a). (c) Atomic changes for the example program, with their interdependences. (d) Call graphs for the tests in the original program version in (a). (e) Call graphs for the tests in the edited program version in (a). (f) Intermediate program P_1 after applying atomic change 4 to the original program. (g) Intermediate program P_2 after applying atomic changes 9, 10 to the original program.

change 4, which is dependent on atomic change 1. Node `C.bar()` corresponds to atomic change 10, which is dependent on atomic changes 6 and 9. Finally, the edge labeled $\langle$`C,A.bar()`$\rangle$ corresponds to atomic change 11, which is also dependent on atomic change 9. Consequently, `test3` is affected by atomic changes 1, 4, 6, 9, 10 and 11. Similarly, `test2` is affected by atomic changes 1 and 4.

2.2 Exploring Changes Using Crisp

The original program passed all the tests, but the edited version failed `test3`. As Fig. 1c shows, there are 12 atomic changes for the entire program and 6 of them are considered affecting changes for `test3`. The question is, *which of those six changes are the likely reason(s) for the test failure?* Our tool *Crisp* helps programmers locate the failure-inducing changes by performing automatic construction of valid intermediate program versions containing programmer-specified atomic changes. To a first approximation, *Crisp* works as follows: From the set of affecting changes of a failed test, a programmer may guess the likely reason(s) for the test failure and select those suspected atomic changes. Then, *Crisp* calculates the *to-be-applied set*, which contains the selected atomic changes as well as all the other necessary atomic changes, to generate a valid intermediate program version by adding the set to the original program version.

For `test3`, a programmer may first guess that the change to method `B.foo()` is the reason for its failure. When the programmer selects atomic change 4, following the dependence graph shown in Fig. 1c, *Crisp* automatically augments the to-be-applied set to include atomic change 1 (i.e., $AM(B.inc())$). In addition, *Crisp* always extends an **AM** change by adding the corresponding **CM** change for the same method (if there is such a **CM** change) to generate the valid intermediate version. So atomic change 2 (i.e., $CM(B.inc())$) and its prerequisite atomic change 3 (i.e., $AF(A.x)$) are also added to the to-be-applied set. Thus, selecting atomic change 4 results in *Crisp* applying atomic changes 1, 2, 3, and 4 to create the intermediate program P_1 shown in Fig. 1f. Notice that the affecting changes set of `test3` does not include atomic changes 2 and 3, but *Crisp* augments these two changes into the to-be-applied set. Fortunately, these augmented changes do not affect the test. (More details are discussed in Section 4.1.)

Note that programmers can select any affecting changes they want to inspect in any order. *Chianti* provides the dependences among atomic changes and *Crisp* automatically forms the to-be-applied set and generates the intermediate program, which is independent of the program's development history.

The programmer can now execute `test3` against P_1 and find that it succeeds. The programmer may then suspect that the newly added method `C.bar()` is the potential culprit in the edit. Having created P_1 does not limit the programmer's choice for selecting the next affecting change to inspect. *Crisp* keeps track of a running list of affecting changes that have already been applied to the original program. Any additional affecting changes (and their prerequisites) that the programmer selects are compared to this list to ensure that changes are applied once and only once.

On the other hand, programmers are provided with a rollback function that allows them to undo their selections, restore the original program, and begin exploration again. Suppose that we restart from the beginning and obtain another intermediate version P_2 shown in Fig. 1g by applying atomic change 9. Re-executing `test3` on P_2 results in a failure, revealing that atomic changes {9, 10, 11} are failure-inducing changes. Then, the programmer can work in the intermediate version, which includes fewer atomic changes than the edited version, and focus on method `C.bar()` to find the exact reason that makes `test3` fail. Note that since atomic change 11 is a consequence of applying atomic change 9, it is also considered as a failure-inducing change.

With the help of *Crisp* (and *Chianti*), a programmer can effectively pinpoint the 3 failure-inducing changes out of the 12 atomic changes in the edit. For large applications where the edited version contains thousands of atomic changes, the benefits of having tools to assist in the analysis and to locate relevant changes are undeniable.

3 Atomic Changes and Their Dependences

Chianti is a change impact analysis tool for Java that is implemented in the context of the *Eclipse* environment. Given two versions of a Java program, *Chianti* first performs a pairwise comparison of their abstract syntax trees and decomposes the source code modifications into a set of interdependent atomic changes A, whose granularity is roughly at method level. *Chianti* handles the full Java language (J2SE 1.4). In addition to the changes defined in Section 2, *Chianti* also defines atomic changes for changing an instance field initializer (**CFI**) or a static field initializer (**CSFI**), adding, changing and deleting an instance initializer of a class (**AI**/**CI**/**DI**), and adding, changing, and deleting a static initializer of a class (**ASI**/**CSI**/**DSI**).

Atomic changes have syntactic interdependences which induce a partial ordering $\prec$ on a set of them, with transitive closure $\preceq^*$. $C_1 \preceq C_2$ denotes that C_1 is a prerequisite for C_2. *Crisp* relies on the automated computation of underlying interdependences between atomic changes by *Chianti* to generate the intermediate program from user-specified atomic changes. The dependences defined in this section are syntactic dependences that must be satisfied to ensure compilability. We also define *mapping dependences*, which are used to correlate all other kinds of changes to method-level changes so that *Chianti* can calculate the affected tests and affecting changes correctly.

In *Chianti*, three kinds of dependences are defined between atomic changes and we will discuss them one by one in the following sections.

3.1 Structural Dependence

Intuitively, *structural dependences* capture the necessary sequences that occur when new Java elements are added or deleted in a program.

Adding/deleting Java elements. In our definition, all the adding changes (**AC**, **AF**, **AM**, **AI**, and **ASI**) and deleting changes (**DC**, **DF**, **DM**, **DI**, and **DSI**) represent adding or deleting an empty element.[4] Generally, a new program element must be declared before making any changes to its body. Similarly, the program element body must be cleared before deleting the element itself. For example, if a programmer adds a new class `C` with some fields, methods, and member classes defined, then $AC(C)$ is the structural prerequisite of all the **AF**s, **AM**s, and **AC**s inside class `C`. Similarly, a field must be added before making any changes to its field initializer and a method or an initializer must be declared before making any changes to its body blocks. The dependence between atomic change 1 and atomic change 2

4. **AC** and **DC** both represent changes to classes and interfaces.

```
abstract class A {
  abstract void foo();

}

class B extends A {
  public void foo(){ }
}
```

(a)

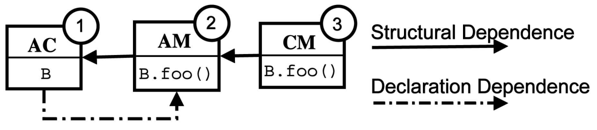
```
abstract class A {
  void foo()
      {// Do Something;}
}
class B extends A {
}
```

(b)

Fig. 2. Changing abstract method declarations and implementations. (a) Original program P, the code to be deleted is underlined. (b) Edited program P', the new added code is shown in boxes.

```
abstract class A {
  abstract public void foo();
}
class B extends A{  1
  public void foo(){ }  2,3
}
```

(a)



(b)

Fig. 3. (a) Original and edited version of example program. The added method is shown in a box. Each box is labeled with the numbers of the corresponding atomic changes. (b) Atomic changes and their dependences.

in Fig. 1c is a trivial example of structural dependence, represented as $AM(B.inc()) \prec_{structural} CM(B.inc())$.

By splitting the addition of the specification of an element from its implementation, we allow *Chianti* to capture the minimal set of affecting changes. For example, the affecting changes set of `test2` in Fig. 1 only includes $CM(B.foo())$ and its prerequisite $AM(B.inc())$, but not $CM(B.inc())$, since method `B.inc()` is not a node in the call graph of `test2` in the edited version.

Additional examples of structural dependences are the adding and deleting of anonymous classes and local inner classes, which are usually defined inside a block (e.g., anonymous classes can be defined in the initializer of a field). The enclosing element must be declared before adding the anonymous classes or local inner classes. For example, if we add a new method `C.foo()` and define a local class `LocalC` inside its body, *Chianti* reports a structural dependence

$$AM(C.foo()) \prec_{structural} AC(C\$1\$LocalC).$$

Changing definitions of a field or method. During software evolution, programmers may change the type of a field, for example, from `List` to `Map`. *Chianti* decomposes this kind of change into a delete field change (**DF**), an add new field change (**AF**), and a corresponding field initializer change (**CFI**) (if the field has an initializer). *Chianti* also reports a structural dependence: $DF \prec_{structural} AF$. If a programmer wants to apply **AF** to the original program, the corresponding **DF** also must be applied, thus guaranteeing that, in the intermediate version of the program, there is no duplicate field defined. Similar dependences are reported when the return type of a method is changed or an existing interface changes to a class or *vice versa*.

3.2 Declaration Dependence

Generally speaking, *declaration dependences* capture all the necessary Java element declarations that are required to create a valid intermediate version. A simple example is the dependence between atomic changes 1 and 4 in Fig. 1c, represented as $AM(B.inc()) \prec_{declaration} CM(B.foo())$, which means method `B.foo()` requires the declaration of method `B.inc()` in order to compile in the edited version.

Declaration-Usage of Java elements. A program element must be declared before it is used. Similarly, a program element can only be deleted when there is no longer any reference to it. In Fig. 1c, the dependences between pairs of atomic changes (1, 4), (3, 2), (5, 8), (6, 8), and (6, 10) all are declaration dependences. Other examples of declaration dependence include $AC(A) \prec_{declaration} AC(B)$, if type `B` is a subclass of class `A`; $AC(A) \prec_{declaration} AM(X.foo())$, if type `A` is used as the return type or any parameter type of method `X.foo()`; and $AC(A) \prec_{declaration} CM(X.foo())$, if type `A` is used in the changed method `X.foo()` in the edited version.

Abstract method declarations and implementations. Another kind of declaration dependence is related to abstract methods. A method declared *abstract* must be implemented in all the subclasses of an abstract class/interface. For example, assume the original program P defines an interface `I` and class `A` implementing `I`. In the edited program P', we add a new declaration of abstract method `foo()` into interface `I`, and class `A` provides the implementation of this method `foo()`. *Chianti* reports that $AM(A.foo()) \prec_{declaration} AM(I.foo())$, which means that the declaration of new method `I.foo()` depends on the method's implementations in all of its subclasses. Otherwise, adding only method `I.foo()` to the original program P will result in an intermediate program P'' which cannot compile.

A similar dependence occurs when the programmer changes the modifier of a method from *abstract* to nonabstract or *vice versa*. Consider the example program in Fig. 2. The original program P consists of an abstract class `A` declaring an abstract method `foo()` and its subclass `B` implementing method `foo()`. In the edited program P', method `foo()` is deleted from class `B`, and the abstract method `foo()` is changed to a concrete method in class `A`. *Chianti* will report a declaration dependence: $CM(A.foo()) \prec_{declaration} DM(B.foo())$, which means that if the programmer wants to delete the declaration of method `B.foo()`, $CM(A.foo())$ must also be applied, so that in the intermediate version of the program, method `foo()` is actually implemented in class `A`; otherwise, class `B` cannot compile since it extends class `A` but does not implement the abstract method defined in class `A`.

Necessary method declarations for a class. Usually we can declare an empty class without any members, but not always. In some cases, we must add some necessary methods to make the class compile.

One case is when programmers declare a class which extends an abstract class or implements an interface, this new class cannot be empty, but must have some overriding methods implemented. Consider the example program in Fig. 3a (the added code is shown in boxes). The original program consists of an abstract class `A` and the edited program declares a new class `B` which extends class `A` and class `B` overrides method `A.foo()`. As we discussed in Section 3.1, two structural dependences are reported: $AC(B) \prec_{structural} AM(B.foo()) \prec_{strucural} CM(B.foo())$ However, these dependences alone are not sufficient to guarantee a valid intermediate program. For example, if only $AC(B)$ is selected to apply to the original program, then we get an intermediate program P'' that defines an empty class `B` which cannot compile since no overriding method `foo()` is defined. Thus, *Chianti* also reports a declaration dependence: $AM(B.foo()) \prec_{declaration} AC(B)$. Fig. 3b shows the dependence graph between the atomic changes. We observe that the structural dependences and declaration dependences between atomic changes 1 and 2 form a cycle, which means that these atomic changes are not separable, and must always be applied together to create a valid intermediate program.

A similar declaration dependence occurs when the programmer declares a class `B` which extends a superclass `A` that defines constructors with arguments but no default (no-argument) constructor. Class `B` must define some constructor which calls method `super(..)` explicitly. In such cases, *Chianti* reports a declaration dependence $AM(B.B(\ldots)) \prec_{declaration} AC(B)$ which forms a cycle with the structural dependence

$$AC(B) \prec_{structural} AM(B.B(\ldots));$$

this forces the intermediate program to always include the corresponding constructors when the class is added and guarantees the compilability of the intermediate program.

3.3 Mapping Dependence

The dependences we have discussed so far are all explicit syntactic dependences that are necessary to build the valid intermediate program. In contrast, a mapping dependence is an *implicit* dependence that is introduced by our atomic change model. As we showed in Section 2, our analysis is at method level. Recall that, to obtain the affecting changes for a given affected test, *Chianti* constructs the test's call graph in the edited program and checks for changed methods (**CM**) that correspond to a node in the call graph and lookup changes (**LC**) that correspond to an edge in the call graph. Therefore, all the other kinds of changes need to map to method changes (**CM**) or lookup changes (**LC**); this gives rise to *mapping dependences*.

Field/initializer changes. In a Java program, changes to initializer blocks and field initializers have repercussions for the constructor or static initializer method of a class. Specifically, if changes are made to instance field initializers or to instance initializer blocks of a class `C`, then *Chianti* also reports a **CM** for each of class `C`'s explicitly defined constructors or reports a **CM** for the implicitly declared default constructor $\texttt{C}.\langle\texttt{init}\rangle()$ and builds mapping dependences between (field) initializer changes and the corresponding **CM** changes. Similarly, if changes are made to *static* initialization blocks (**CSI**) or class variables (**CSFI**) of class `C`, then mapping dependences are created between **CSI** or **CSFI** changes and method change $CM(C.\langle clinit\rangle())$.

Overloading methods. Overloading poses interesting issues for change impact analysis. Consider the introduction of an overloaded method as shown in Fig. 4 (the added method is shown in a box). Note that there are no textual edits in `R.bar()`, and further, that there are no **LC** changes because all the methods are `static`. However, adding method `R.foo(Y)` changes the behavior of the program

```
class R {
  static void foo(X x){ }
  static void foo(Y y){ }
  static void bar(){
    Y y = new Y();
    R.foo(y);
  }
}
class X { }
class Y extends X { }
class Test extends TestCase{
  public void testBar(){
    R.bar();
  }
}
```

Fig. 4. Addition of an overloaded method. The added method is shown in a box.

because the call of `R.foo(y)` in `R.bar()` resolves (after the change) to `R.foo(Y)` instead of `R.foo(X)` [7], and affects the call graph of `Test.testBar()`. Therefore, *Chianti* reports a **CM** change for method `R.bar()` despite the fact that no textual changes occur within this method,[5] and creates a mapping dependence:

$$AM(R.foo(Y)) \prec_{mapping} CM(R.bar()).$$

Field type or method return type changes. As we discussed in Section 3.1, if the programmer changes the type of a field, *Chianti* reports two atomic changes, **DF** and **AF**, and a structural dependence, $DF \prec_{structural} AF$. In addition, *Chianti* also searches for all the methods, initializers, and field initializers that refer to this changed field, reports those as changed elements, and creates the corresponding mapping dependences. Considering the example program in Fig. 5, in the original program, class `A` declares two fields `int i;` and `int j;`, method `foo()` does some operations on these two fields, and `testA()` calls method `A.foo()`. Running `testA()` on the original program prints `0` as the output. In the edited program, the type of fields `i` and `j` are both changed to `String`. The output of `testA()` changes to `null` instead of `0` in the edited program. *Chianti* searches for the uses of fields `i` and `j` in the original program, which is method `A.foo()` in this example. Then *Chianti* reports a **CM** change to this method and creates mapping dependences

$$\begin{aligned} &\{AF(A.i), AF(A.j)\} \prec_{mapping} CM(\texttt{A.foo}()) \text{ and} \\ &\qquad CM(\texttt{A.foo}()) \prec_{mapping} \{DF(A.i), DF(A.j)\}, \end{aligned}$$

shown in Fig. 5. Notice that these dependences form two dependence cycles, thus forcing all five changes to be applied together to obtain the intermediate version, no matter which of the changes is selected by the programmer. Otherwise, if no mapping dependences are reported and the programmer wants to apply $AF(A.i)$ to the original program, then the resulting intermediate program will report a compilation error for method `A.foo()`, since a `String` cannot be assigned to an `int`.

LC changes. LC changes model changes to the dynamic dispatch behavior of instance methods. In particular, $LC(Y, X.m())$ models the fact that a call to method `X.m()` on an object of runtime type `Y` results in the selection of a

5. However, the abstract syntax tree for `R.bar()` will be different after applying the edit, as overloading is resolved at compile time.

```
class A {
  int i, j;
  public void foo(){
    j = i;
    ...println(j);
  }
}
```

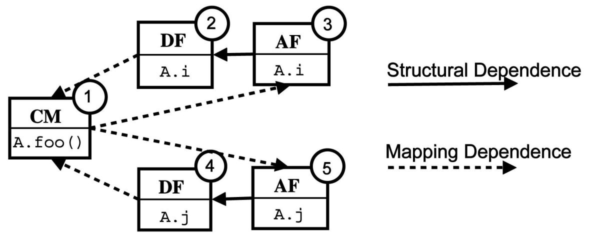
(a)

```
class A {
  String i, j;
  public void foo(){
    j = i;
    ...println(j);
  }
}
```

(b)

```
class Test extends TestCase {
  public void testA(){
    new A().foo();
  }
}
```

(c)



(d)

Fig. 5. Field type changes. (a) Original program P. (b) Edited program P'. (c) The test used in both P and P'. (d) The dependence graph between changes.

different method in the edited program. Many source code changes can be mapped to **LC** changes.

Addition or deletion of a class may result in **LC**. Considering the example in Fig. 1a, if the programmer adds "`class D extends A`" with empty members in the edited program P'. This edit results in a list of **LC** changes (e.g., $LC(D, A.foo())$, $LC(D, A.bar())$) since in the original program P, there is no dynamic lookup for runtime type `D`, while in the edited program P', programmers can invoke a call to method `A.foo()` on an object of type `D`. *Chianti* reports a mapping dependence:[6]

$$AC(D) \prec_{mapping} LC(D, A.foo()).$$

The addition or deletion of an overriding method may also result in **LC** changes. Consider the example in Fig. 1a, a new overriding method `bar()` is added to class `C` in the edited program P'. Two mapping dependences are shown in Fig. 1c:

$$AM(C.bar()) \prec_{mapping} LC(C,\ A.bar()) \text{ and}$$
$$AM(C.bar()) \prec_{mapping} LC(C,\ C.bar()).$$

Changing the access control of a method or changing the modifier of a class are other sources of **LC** changes. For example, a *private* method is not dynamically dispatched, but if we change the method's modifier to *public*, then an entry for this method must be added in the new dynamic dispatch map and a mapping dependence between the **CM** and **LC** changes is created. Similarly, making an *abstract* class `C` nonabstract also results in **LC** changes. In the original dynamic dispatch map, there is no entry with `C` as the runtime receiver type, but the new dispatch map will contain such an entry.

6. *Chianti* also reports **LC** changes for other inherited methods from library classes, for example,
$AC(D) \prec_{mapping} LC(D, java.lang.Object.toString())$

4 Constructing Intermediate Program Versions

Regression test suites are usually executed before committing an edit of an application into a version control repository. When a regression test fails after a long editing session, locating those changes that correspond to the failure is not a trivial task. The affecting changes generated by *Chianti* for each affected test are the first step in isolating the changes that potentially cause the failure of a test.

The affecting changes of a failed affected test can be applied incrementally to the original program to obtain an intermediate program P_1. A programmer can then re-execute the affected test against P_1 and, based on the outcome of the test, isolate the few changes that potentially cause the failure of the test. As a companion tool to *Chianti*, we developed *Crisp* to create intermediate program versions based on programmer selections.

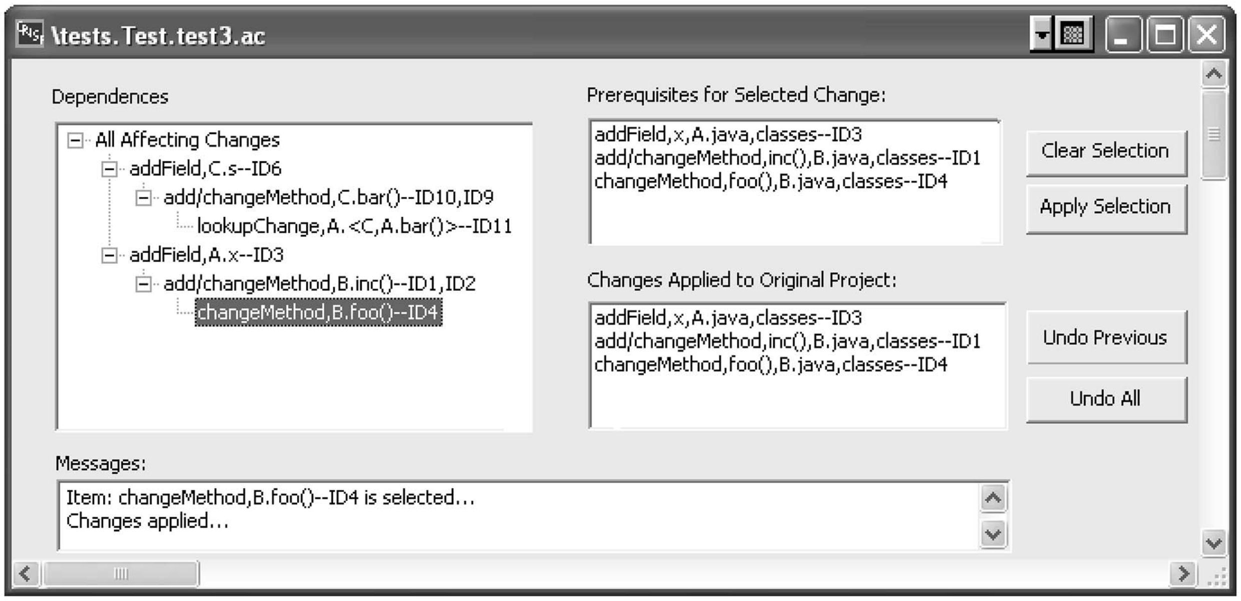
Like *Chianti*, *Crisp* is built as an Eclipse plug-in. *Crisp* takes as input the atomic changes generated by *Chianti* for two versions of a Java program, as well as the affecting changes of an affected test. The major tasks that *Crisp* performs are 1) to gather and order all the prerequisites of the affecting changes and to create their to-be-applied sets, 2) to present the to-be-applied sets in a dependence tree format, and 3) to respond to a programmer's selection of an affecting change(s) and automatically build a syntactically correct intermediate program P_1.

The Eclipse Plug-in Development [3] environment provides APIs for accessing the abstract syntax trees of the original and the new Java programs and for programmatically manipulating the source code of Java class files. The abstract syntax trees contain source locations of Java constructs and therefore ease the effort of pinpointing the locations of all of these affecting changes. In order to accomplish the task of creating syntactically correct versions of program, there are several practical aspects of how *Crisp* uses atomic changes that differ from their original usage in *Chianti*.

4.1 Creating the To-Be-Applied Change Set

The ultimate goal in *Crisp* is to create the corresponding to-be-applied set for each affecting change that the programmer selects. The three categories of dependences defined in Section 3 are used by *Crisp* to collect all the necessary prerequisites in the to-be-applied set. For declaration and mapping dependences, the ordering of the changes in the to-be-applied set is arbitrary. In Fig. 1, whether we add the method `B.inc()` before or after changing method `B.foo()` does not matter, (i.e., these changes are related by a declaration dependence), as long as both of these changes are present in the final intermediate version to be compiled. However, certain orderings of structural dependences among affecting changes are critical to the process of *Crisp* creating a valid intermediate version. To-be-applied sets are, therefore, partially ordered. Within *Crisp*, structural dependences can be divided into two categories: 1) *pure structural* dependences and 2) *buddy* dependences.

Pure Structural Dependences. Most of the structural dependences belong to this category and *Crisp* must handle


\tests.Test.test3.ac

Dependences

```
All Affecting Changes
  addField,C.s--ID6
    add/changeMethod,C.bar()--ID10,ID9
      lookupChange,A.<C,A.bar()>--ID11
  addField,A.x--ID3
    add/changeMethod,B.inc()--ID1,ID2
      changeMethod,B.foo()--ID4
```

Prerequisites for Selected Change:

```
addField,x,A.java,classes--ID3
add/changeMethod,inc(),B.java,classes--ID1
changeMethod,foo(),B.java,classes--ID4
```

Changes Applied to Original Project:

```
addField,x,A.java,classes--ID3
add/changeMethod,inc(),B.java,classes--ID1
changeMethod,foo(),B.java,classes--ID4
```

Messages:

```
Item: changeMethod,B.foo()--ID4 is selected...
Changes applied...
```

Fig. 6. Screen shot from *Crisp*.

the corresponding changes in order, when constructing the intermediate version. For example, a member method or a member field cannot be added or edited unless its newly added enclosing class has been added. Similarly, a field needs to be deleted prior to the addition of a field with the same name, yet different type.

Buddy Dependences. The granularity of the output of *Chianti* is at the atomic change level; however, certain atomic changes by themselves do not constitute *legitimate* program edits. For example, *Chianti* distinguishes between an **AM** change and a **CM** change to a newly added method in the edited version. *Chianti* also creates a structural dependence between them indicating that the **AM** declaring the method, should be applied before the **CM** creating the method body. However, in the context of *Crisp*, presenting the **AM** and the **CM** may be confusing to programmers. Adding the method signature without its body very often results in a syntax error due to a missing return statement. Furthermore, there is no compelling reason to test the reexecution of empty methods. Similar circumstances apply to other changes as well. A programmer who writes `String s = ''abc''` (identified by *Chianti* as an **AF** change and a **CFI** change) probably will not be interested in testing the program that only declares the field (**AF**). We therefore aggregate the following atomic changes into *buddy pairs* based on their semantics:

1. add/changeMethod—add the method declaration and body,
2. change/deleteMethod—delete the method body and declaration,
3. add/changeFieldInitializer—add a field variable, its type, and its initial value, and
4. change/deleteFieldInitializer—delete a field variable, its type, and its initial value.

Intuitively, the atomic changes in a buddy pair are not only ordered, they are *inseparable*. There are also buddy pairs for initializers and static initializers, but they are not as common (i.e., in our experience) as those we listed here.

With buddy pairs, programmers will be able to select a change of interest, without needing to understand the technical difference between an **AM** versus a **CM**, when adding a new method to the original program. *Crisp* combines the prerequisites of the two individual atomic changes in the buddy pair and applies all of them at the same time. This process, though necessary for the reasons given above, adds changes to the to-be-applied set that are not necessarily affecting changes themselves (with respect to the test being investigated). In Section 2, Fig. 1f shows the result of the programmer selecting $CM(B.foo())$. $CM(B.foo())$ has a prerequisite $AM(B.inc())$. *Crisp* combines $AM(B.inc())$ and $CM(B.inc())$ into a buddy pair. Finally, $AF(A.x)$ is included in the to-be applied set because it is a prerequisite for $CM(B.inc())$, even though it is not an affecting change for `test3`.

4.2 Local and Anonymous Classes

The notion of combining atomic changes and presenting them to the programmers as buddy pairs is a way of taking into account the level of granularity we envision for *Crisp* usage. Most regression tests focus on functionality at a method-level instead of the inner details of method implementation that are (supposedly) tested during the unit testing phase. On the other hand, the Java language itself provides numerous constructs below the method-level such as local and anonymous classes. *Chianti*, being comprehensive and adhering to the Java specification, also analyzes these sub-method-level atomic changes. However, the Java naming convention for an anonymous class complicates pinpointing its relative position within the enclosing method [1] in creating the intermediate program versions. The current implementation of *Crisp* therefore, does not handle sub-method-level changes individually, but applies all of them as a **CM** change. We may consider alternative handling of such changes in our future work.

4.3 Crisp Interface

Crisp presents a graphical depiction of a dependence tree listing all the affecting changes, the buddy pairs, and their relationship with their prerequisites. For each of these items, *Crisp* maintains a lookup to its corresponding to-be-applied set. Fig. 6 shows a screen shot of *Crisp*. Using Fig. 1 in Section 2, atomic change 3 $AF(A.x)$ is listed as a parent

TABLE 1
The Sizes of Case Study Data

Version Pairs	KLOC	Classes	Methods	Tests
Daikon Nov 11–19, 2002	78	581	6,017	62
Eclipse jdt compiler 2003	155	841	10,154	1,477
Eclipse jdt compiler 2004	191	965	12,834	5,023

of buddy pair $AM(B.inc())$ and $CM(B.inc())$ (atomic changes 1 and 2, which can only be selected as a pair as explained previously), indicating that adding the field `A.x` is a prerequisite to atomic changes 1 and 2, which in turn are prerequisites to atomic change 4, $CM(B.foo())$.

5 Case Studies

It was a challenge to identify appropriate test data for *Crisp* obtainable from a real-world software project outside of our research group. For our purposes, we must have access to both the software source code and a comprehensive regression test suite that must contain some failing tests. In addition, since bugs in small programs can be found using traditional tools such as the GUI debugger in Eclipse, our techniques should be applied to software of at least moderate size. In this section, we present two case studies using CVS repository data from *Daikon* [2] and *Eclipse jdt compiler* [3]. Table 1 shows a summary of them. We use eight version pairs from *Eclipse jdt compiler* and separate them into two parts, one for 2003 and one for 2004. The numbers shown in the table are average numbers over the version pairs in the period. The KLOC is the number of thousands of lines of *uncommented* code.

5.1 Daikon Unit Tests

In previous research [1], we had extracted 52 weeks of Daikon [2] source code from a CVS repository for testing the effectiveness of *Chianti* (i.e., all year 2002 check-ins). We were not surprised to find that executing these tests did not result in any failures because, commonly, failed tests delay code check-in until the source of the problem is fixed. In order to simulate the development life cycle of Daikon, we chose a version pair and attempted to execute test suite version n against source code version $n+1$. This mimicked the situation where the editing of the new version of the source code was complete and the programmer was ready to execute the existing test suite on the new code.

We found two tests, `testMinus` and `testXor` in package `daikon.test.diff`, that executed successfully in the 11 November version, but failed in the edited version dated 19 November. We therefore used *Chianti* to generate the atomic changes for these two versions, to confirm that the two tests are in fact affected, and to calculate their affecting changes. The results were then passed to *Crisp* to create the to-be-applied sets and present them in the dependence tree window. Our goal was to use *Crisp* to locate the changes that had caused the failure of these tests.

The results are summarized in Table 2. *Chianti* calculates 6,093 atomic changes between these two versions of Daikon. For `testXor`, there are 35 affecting changes and using *Crisp* we found two failure-inducing changes. Similarly, there are 34 affecting changes for `testMinus` and with *Crisp*, we located one failure-inducing change. For both tests, the to-be-applied sets are exactly the same as the affecting changes set.

TABLE 2
Using *Crisp* for the Daikon Versions 11 November and 19 November 2002

Atomic changes	Failing tests	Affecting changes	Changes explored	Failure-inducing changes
6093	testXor	35	20	2
	testMinus	34	13	1

Since we were not familiar with the source code of Daikon, we attempted to locate the changes that caused test failure in a naive manner. The fact that there were only 34 or 35 affecting changes for each test made it simple to derive an approach. We added changes with no prerequisites first, then those with one prerequisite, etc. During this process, we rolled back to the original program after each change to apply the next one.

Selecting changes without prerequisites turned out to be extremely effective in this case study. For test `testMinus`, we were able to locate an atomic change *CM(daikon.diff.MinusVisitor.shouldAdd(..))* whose application to the original program caused failure. For test `testXor`, selecting changes without prerequisites resulted in finding atomic change *CM(daikon.diff.XorVisitor.shouldAddInv2(..))* that caused a failure.

In order to confirm our results, we applied all the changes except *CM(MinusVisitor.shouldAdd(..))* to the original program and reexecuted test `testMinus`, which then succeeded. This showed that *CM(MinusVisitor.shouldAdd(..))* was the only *failure-inducing change* for test `testMinus`. However, for test `testXor`, the application of the complementary changes set also resulted in test failure. We then continued our approach, selected other changes without prerequisites, and found atomic change *CM(daikon.diff.XorVisitor.shouldAddInv1(..))* to be another failure-inducing change for test `testXor`. Neither *CM(XorVisiteor.shouldAddInv1(..))* nor *CM(..XorVisitor.shouldAdInv2(..))* has prerequisites and they are independent of each other. Checking the complementary changes without these two atomic changes confirmed that these are the only failure-inducing changes for test `testXor`.

5.2 Eclipse jdt Compiler Unit Tests

We performed our next case study on an Eclipse plug-in project, `org.eclipse.jdt.core`, which has several associated test plug-ins. We chose `org.eclipse.jdt.core.tests.compiler` to do the case study because of its availability with the development history of `org.eclipse.jdt.core`.[7] Each nightly build of these two plug-ins from 2003 to 2005 was checked out and considered as a version. We executed all the tests in the packages `parser` and `regression` within the compiler tests plug-in, attempting to find failing tests. We succeeded in finding four version pairs that have successful test results in the original version and failing tests in the next nightly build (edited) version, which are indexed 1 to 4 in Table 3. Then, we applied the same strategy as for the Daikon data by applying version n tests to version $n+1$ source code to induce failing tests, and found four more version pairs in 2003 which are indexed 5 to 8 in Table 3.

7. The other reason is because of the properties of the unit tests. The tests in other three test plug-ins are JUnit plug-in tests, for which we currently cannot build call graphs effectively.

TABLE 3
Using *Crisp* for the Eclipse jdt Compiler Versions with Failing Tests

Index	Version pair	Atomic changes	Failing tests	Avg. affecting changes	Avg. to-be-applied sets	Changes explored	Avg. failure-inducing changes
1	31Mar2003–01Apr2003	370	1	30	58	n/a	*Crisp* failed
2	13Aug2003–14Aug2003	101	3	1	1	1	1
3	10Apr2004–11Apr2004	146	46	37	54	12	1
4	16Nov2004–17Nov2004	465	4	15	19	5	2
5	21Jan2003–22Jan2003	724	7	151	187	n/a	3
6	22Jan2003–23Jan2003	723	2	6	7	5	1
7	14Feb2003–15Feb2003	762	2	11	12	5	2
8	24Jun2003–25Jun2003	156	1	3	3	2	1

TABLE 4
Details of Test Results for Eclipse jdt Compiler Version Pair 21Jan2003-22Jan2003

Index	Tests	# of Affecting Changes	Size of to-be-applied sets	Failure-inducing Changes
1	CompletionParserTest.testZA_1	142	174	CM CompletionParser.consumeEnterVariable()
2	CompletionParserTest.testV_1FGGUOO_1	146	182	AM/CM CompletionParser.consumeClassHeaderName()
3	CompletionParserTest.testZ_1FGPF3D_1	146	182	LC(...jdt.internal.codeassist.complete.CompletionParser,
4	DietCompletionTest.test16	146	182	...jdt.internal.compiler.parser.consumeClassHeaderName)
5	DietCompletionTest.test17	146	182	
6	CompletionParserTest.testDB_1FHSLDR	164	202	CM CompletionParser.createSingleAssistNameReference(char[], long)
7	CompletionParserTest.testHB_1FHSLDR	164	202	CM CompletionParser.consumeToken(int)
				CM AssistParser.popElement(int)

All tests are from package `org.eclipse.jdt.core.tests.compiler.parser`. The failure-inducing changes of tests 1 through 5 and the first two failure-inducing changes of tests 6 and 7 are from package `org.eclipse.jdt.internal.codeassist.complete`. The last failure-inducing change of tests 6 and 7 is from package `org.eclipse.jdt.internal.codeassist.impl`.

TABLE 5
Details of Test Results for Eclipse jdt Compiler Version Pair 16Nov2004-17Nov2004

Index	Tests	# of Affecting Changes	Size of to-be-applied sets	Failure-inducing Changes
1	CompilianceDiagnoseTest.test0046	22	26	CM Parser.consumeAnnotationTypeMemberDeclaration()
				CM Parser.consumeAnnotationTypeMemberHeaderExtendedDims()
				CM Parser.consumeAnnotationTypeMemberDeclarationHeader()
				CM ClassScope.connectSuperInterfaces()
2	ComplianceDiagnoseTest.test0047	13	17	
3	Compliance_1_3.test089	13	17	CM ClassScope.connectSuperInterfaces()
4	Compliance_1_4.test089	13	17	

*Tests 1 and 2 are from package `org.eclipse.jdt.core.tests.compiler.parser`. Tests 3 and 4 are from the package `org.eclipse.jdt.core.tests.compiler.regression`. The first three **CM**s of test 1 are from package `org.eclipse.jdt.internal.compiler.parser`. (The first three **CM**s when applied individually to a test, result in exceptions; we consider these to be failure inducing changes for this study.) The other **CM**s are from package `org.eclipse.jdt.internal.compiler.lookup`.*

Since most of these version pairs contain more than one failing test, each with various numbers of affecting changes, we provide the averages over all failing tests per version pair in Table 3. In addition, we selected two version pairs, for which we present further details of our findings in Table 4 and Table 5. The size of the to-be-applied column in these tables indicates the number of changes that are presented in the dependence tree. This number is very often higher than the number of affecting changes because in some call graphs, only an **AM** is in the affecting changes set, but *crisp* needs to augment the to-be-applied set to include its buddy changes and recursively all the corresponding prerequisites.

For this case study, we used a different exploration strategy because for some of the tests, the affecting changes are more interdependent, (i.e., many of them have prerequisites and some of the dependences are more than five levels deep). The naive approach of choosing changes without prerequisites turned out to be not as effective as with the Daikon data. For the version pair `21Jan2003 – 22Jan2003` in particular, only one **CM** change has no prerequisites, and it is not the failure-inducing change. All the changes that have no prerequisites are either **AC** or **AF** changes. From a semantic point of view, choosing **AC** or **AF** changes is not useful since they cannot be failure-inducing changes by themselves. On the other hand, choosing **CM** changes that have more than three to four levels of dependences is not effective either. The latter changes usually have a large number of prerequisites and if applying such changes results in the test failure, it does not help too much in locating the small number of failure-inducing changes.

We therefore devised another strategy. The changes we first selected were **CM** changes that had only class and/or field changes as their prerequisites. Once we had added these changes and the test results had remained successful, we then began to add **CM** changes and **LC** changes that were one level deeper in the dependence relationship. This way, each application of selected changes contained only a few prerequisites which were easy to track. Using this strategy for the `21Jan2003 - 22Jan2003` version pair, we located the failure-inducing changes easily for the first five tests shown in Table 4. The first test failed because of *CM(CompletionParser.consumeEnterVariable())*, which has an **AM** change as its prerequisite. Tests indexed 2 to 5 share the same failure-inducing changes, which are from a newly added method, *CompletionParser.consumeClassHeaderName()*, with another six **AC**, **AF**, and **AM** changes as their prerequisites. For the last two tests, three **CM** changes combined together make the tests fail, and applying any subset of these three changes results in tests that do not fail. In total, these three **CM** changes have 55 **AC**, **AF**, and **AM** changes as their prerequisites.

Note that in Table 4, the failure-inducing changes for tests indexed 2 to 5 are newly added methods, not changes to existing methods. This implies that the addition of these methods alters the dynamic dispatch behavior of the execution of the test. We confirmed that *CompletionParser. consumeClassHeaderName()* does have a mapping dependent **LC** change which is also in the affecting changes set of these tests.

In the case study, we also noticed that many of these failure-inducing changes cause failures in multiple tests which have similar affecting changes sets. In Table 4, *AM(CompletionParser.consumeClassHeaderName())* causes four tests to fail. In Table 5, the atomic change *CM(ClassScope.connectSuperInterfaces())* causes the failure of all four failing tests in the version pair. This suggests that certain changes fall within the execution paths of related tests, and the timely identification of specific failure-inducing changes could have a positive impact on the development time frame.

In all but one case in these eight versions, with a total of 66 failing tests, we were able to use *Crisp* to locate the failure-inducing changes. In version pair `31Mar2003 - 01Apr2003`, we experienced a limitation of *Crisp*. A **CM** change defined inside an anonymous class is introduced by *Crisp* while building the to-be-applied set. Since *Crisp* can not edit an anonymous class within a method, it failed to build a compilable version, thus revealing a limitation. This failure was further complicated by a change to the class hierarchy in the edited version (which will be discussed in detail in Section 6).

5.3 Discussion of the Two Case Studies

From these two case studies, we demonstrated the potential use of *Crisp* in assisting programmers to explore and locate failure-inducing changes for a regression test. On average, 7 percent of the atomic changes are affecting changes for each test; using *Crisp*, we further isolate the failure-inducing changes to one to four changes.

A programmer can use *Crisp* to apply only a small subset of suspicious changes to the original program. Even with interdependences among affecting changes, we have demonstrated two major strategies for exploration that keep the number of prerequisites manageable for each selection of changes. Table 2 and Table 3 include the average number of affecting changes explored to locate the failure-inducing changes. Note that the data provided is based on our heuristic except for version pair 5 in Table 3, for which we combined several heuristics to quickly locate the failure-inducing changes. We are currently working on finding more effective heuristics to improve the exploration. The ability to roll back the changes also allows a programmer to explore in free form—changes can be aggregated in different ways and each exploration can be performed independently. Rollback was used extensively in our exploration in applying the complement set of the failure-inducing changes.

In addition, for more coarse-grained exploration, the programmer may choose to consider several related affected tests together with their set of affecting (and possibly interacting) changes. This allows exploration of intermediate program versions corresponding to the changes from multiple tests.

The performance of *Chianti* and *Crisp* has thus far not been our primary focus; however, we have achieved acceptable performance for a prototype. On average, deriving atomic changes for two successive versions takes 24 milliseconds per method. Given the call graphs for each test, determining the affected tests and affecting changes takes only a few seconds. It takes *Crisp* less than a second on average to obtain the to-be-applied set and apply it to the original program for each selected change. All measurements were taken on a Pentium 4 PC at 2.8 Ghz with 1 Gb RAM.

TABLE 6
The Comparison of the Average Numbers of Failure-Inducing Changes of Two Definitions on Eclipse jdt Compiler

Version Pair Index	1	2	3	4	5	6	7	8
Optimistic	Fail	1	1	2	3	1	2	1
Pessimistic	Fail	1	2	5	22	1	3	1

5.4 Defining a Failure-Inducing Change

In the case study, we use the following definition to identify failure-inducing changes:

Let A be the affecting changes for a given test t. Form the smallest subset $A' \subseteq A$, such that applying A' to the original program P results in an intermediate program P_1 on which test t fails, and applying all the changes in $A - A'$ to the original program results in intermediate program P_2 on which test t does not fail. Then, all the changes in A' are *failure-inducing changes*.

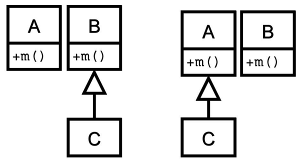
We refer to this definition as *optimistic*, because it only "counts" the changes that actually make a test fail, not including their prerequisites. The figures in Tables 2 and 3 were derived using this optimistic definition. In our case study, only **AM**s, **CM**s, and **LC**s are reported as failure-inducing changes.

An alternative *pessimistic* definition is to include all the transitively prerequisite changes derived from A' as failure-inducing changes. For the case study on Daikon, since all the failure-inducing **CM** changes do not have prerequisites, the difference between the optimistic and pessimistic definitions does not affect the data reported. For the case study on the *Eclipse jdt compiler*, many of the failure-inducing **CM** changes have some other **AF**s, **AC**s, and **AM**s as prerequisites; thus, use of the pessimistic definition results in more failure-inducing changes. Table 6 shows the average number of failure-inducing changes that would be reported in the *Eclipse jdt compiler* case study for each of the two definitions. For version pairs 4 and 5, we notice an obvious difference between the two definitions.

It is important to investigate whether or not each prerequisite for a failure-inducing change was *only* a prerequisite for that change or was a prerequisite for other changes as well. If a prerequisite is shared with other

```
class R {
  public void foo( ){
    Object o = new C();
    ...(B)o...
  }
  public void bar(){
    ...
    A a = new C();
    ...
  }
}
```

(a)



(b) (c)

Fig. 7. Type hierarchy change. (a) The program fragments that is related the hierarchy change, new added code is shown in the box. (b) The type hierarchies before edits. (c) The type hierarchy after edits.

```
class A {
  String a = "a";

  String c = a;
}
```

(a)

```
class A {
  String a = "a";
  String b = a;
  String c = b;
}
```

(b)

```
class A {
  String b = a;
  String a = "a";
  String c = b;
}
```

(c)

```
class A {
  String a = "a";
  String c = b;

  String b = a;
```

(d)

Fig. 8. The positions of an added field affects the compilability of a program. (a) The original program P. (b) The edited program P'. (c) The first intermediate version P_1 by putting the new added field at the beginning of a class. (d) The second intermediate version P_2 by putting the new added field at the end of a class.

CM changes, then it is also necessary for them, so we should not classify such a prerequisite as failure-inducing, as this may divert attention from the real failure-inducing change(s). We checked the prerequisites for each failure-inducing **CM** change in these 2 version pairs and found that 5 **AM**s and 31 **AF**s occurred solely as prerequisites of the failure-inducing changes; the other 30 **AC**s, **AF**s, and **AM**s were shared prerequisites with other non-failure-inducing changes. Given the variations of these two definitions, we plan to collect more data and investigate further the root causes of the failures.

6 Limitations of Our Current Implementation

The goal of *Crisp* is to find all the necessary dependences to allow automatic building of intermediate program versions; however, our experiences with the case studies revealed some limitations of our current implementation.

Hierarchy changes. There are no atomic changes defined in *Chianti* to represent hierarchy changes. Instead, we map those changes to the appropriate **LC** changes or **CM** changes. Consider the hierarchy change in Fig. 7b and 7c. *Chianti* reports two **LC** changes to reflect the type hierarchy change of class C: $LC(C, B.m())$ and $LC(C, A.m())$, which represent the fact that in the original program, type C can be used as the runtime type of the call to method B.m(), but not in the edited program, and vice versa.

It is also possible for changes to the hierarchy to affect the behavior of a method, although the code in the method is not changed. For example, in Fig. 7a, method R.foo()} contains a cast to type B. This cast will succeed in the original program since class C is a subtype of B. In contrast, if we make the hierarchy change shown in Fig. 7c, then this cast will fail. Note that the code in method foo() *has not changed* due to the edit, but the behavior of foo() has been altered. *Chianti* maps the hierarchy reconstruction change to **CM** changes for any methods containing a construct that checks the runtime type of the object (i.e., instanceof, casts, and exception catch blocks) for the types involved in the reconstruction. In this example, *Chianti* reports $CM(R.foo())$.

The mapping of type hierarchy reconstruction to **CM** and **LC** changes successfully allows *Chianti* to correctly calculate the affected tests and affecting changes. However, it is insufficient to ensure the construction of valid intermediate programs. Consider the program in Fig. 7. *Chianti* reports $CM(R.bar())$ because of the added code shown in the box. If the programmer wants to apply this **CM** to the original program, *Crisp* will build an intermediate program with the code in Fig. 7a and the type hierarchy in Fig. 7b, *not* Fig. 7c. This intermediate version has a compilation error at the code in the box because in the original type hierarchy, type C is not a subtype of type A. This also was one of the reasons why *Crisp* failed to create a valid intermediate version for version pair 1 in Table 3. Fortunately, we only found one such case in our case studies. We are working currently on defining a new atomic change for these hierarchy changes.

Field positions. The atomic changes and dependences generated by *Chianti* do not include information about the relative position of the change in the code. If an *add* change needs to be applied to the original program, *Crisp* has two choices: adding the new element at the beginning or at the end of the class. In most cases, the position of the elements do not affect the compilation of the program; however, in some end cases, putting fields in the wrong position results in compilation errors in the intermediate program.

Consider the example program in Fig. 8. Figs. 8a and 8b represent the original and edited program, respectively. The added field is shown in a box, and the changed field is underlined. Suppose the programmer wants to apply both changes to new field b and existing field c to the original program. If *Crisp* adds field b at the beginning of the class, we obtain the intermediate program shown in Fig. 8c; it has a compilation error for field b because it refers to field a that is not yet declared. If *Crisp* adds field b at the end of the class, we get the intermediate program shown in Fig. 8d; this also has compilation error for field c because it refers to field b, which is not yet declared.

Crisp always puts new added elements at the end of a class; if this results in an invalid program, the programmer needs to manually change the position of this field in the code. In our case studies, we found one case where a newly added field referred to an existing field; it was safe to append the new field at the end of the class since no other existing field referred to it. Note that in all the tests in *Daikon* and *Eclipse jdt compiler* case studies, the limitation of *Crisp* in locating exact field position in the code did not hinder its effectiveness in locating the failure-inducing changes.

7 Related Work

Our *Chianti* prototype has already been described in [4], [1], [8]. In these papers, we present definitions of atomic changes, affected tests, and affecting changes, described informally here using an example in Section 2. The dependences between atomic changes presented in Section 3 are briefly discussed in [1], [8], but they are not grouped into categories nor explained in the detail given here. This paper is an extension of the work originally published in [5].

Change impact analysis and regression testing are related to the analyses in *Chianti* (used by *Crisp*) and have been discussed extensively in [1]. We summarize those discussions below.

Other areas of research relevant for comparison with *Crisp* are delta debugging, techniques for avoiding recompilation, and fault localization.

7.1 Change Impact Analysis

Other research on impact analysis has concentrated on finding *program constructs* potentially affected by changes. These analyses are based on static analysis [9], [10], [11], [12], dynamic analysis [13] or, like our analysis, on a combination of the two [14]. Recent work on change impact analysis includes the *PathImpact* algorithm by Law and Rothermel [13], 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. [14], which combines the use of a forward static slice [15] 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].

Rajlich et al. [17], [18], [19] propose a methodology to handle incremental change in object-oriented programs. Given a change request, a programmer needs to incorporate the new concept in the code, which may alter existing class dependences. Intuitively, incremental change propagation uses reachability on a class dependence graph to calculate possibly affected classes "downstream" from a code change to the current under edit. In contrast, our change impact analysis aims to find a subset of the changes to a program that impact a test whose behavior has (potentially) changed.

Runtime software evolution is a way to make changes to a software system while it is executing. Gustavsson [20] proposed a classification of runtime software changes to help programmers understanding of such changes. This work addresses a code different change problem than our research.

7.2 Delta Debugging

In the work by Zeller on *delta debugging*, the reason for a program failure is identified as a set of differences between program versions [21] 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, with the set of changes applied, one can narrow down the set of changes responsible for the failure. Delta debugging has been applied successfully to very large programs [21].

In the examination of differences between program versions, both delta debugging and our work aim at identifying failure-inducing changes; however, there are several important differences between the two approaches. First, delta debugging searches the entire set of changes to find the failure-inducing changes. In our approach, we first obtain the set of affecting changes for a failed test with *Chianti*, and we then generate the intermediate versions of programs just from this small set of changes. By associating each test with its corresponding affecting changes, a large set of uncorrelated changes can be ignored, so that a programmer can focus on only those changes related to the given test. This is extremely useful when the reexecution of the regression test suites is costly. Second, delta debugging builds the intermediate versions by using only the structural differences between succeeding and failing program executions (e.g., changing one line or one character to generate an intermediate program version) and it is language-independent. Our model of dependences between atomic changes ensures that *Crisp* only builds meaningful intermediate versions of Java programs, which reduces the number of intermediate programs that need to be constructed. When a programmer selects a set of interesting changes, *Crisp* automatically augments these changes with all the prerequisites necessary to build a syntactically valid program version. Unlike delta debugging, which creates versions automatically, our approach is semiautomatic, requiring programmer selection of the changes to be added. In our future work, we plan to investigate how to extend *Crisp* so it can generate the intermediate versions automatically.

7.3 Fault Localization

Program slicing [15] has been suggested as a technique for localizing faults: Computing a slice with respect to an incorrect value determines all statements that may have contributed to that value, and will generally include the statement(s) that contain the error. Since slices may become very large, techniques such as *program dicing* [22] have been developed, where a slice with respect to an erroneous value is intersected with a slice with respect to a correct value. DeMillo et al. [23] suggest *critical slicing* as a technique to localize faults in a program. A statement is critical if without it, program execution reaches the same failure statement s_F but with different values for referenced variables. They report that their technique is able to reduce relevant program size by around 64 percent and retain the failure-inducing statement in 80 percent of the cases. Bunus and Fritzson [24] suggest a semiautomatic debugging framework for equation-based languages used to model physical systems. Their approach uses program slicing and dicing on a combination of execution traces, dependence graphs, and assertions to help programmers find and correct bugs in an interactive debugging session. Gupta et al. [25] use delta debugging to *simplify* or *isolate* inputs that are failure-inducing and then use forward and backward dynamic slices to suggest a set of statements that could potentially contain the fault.

There are two major differences between our approach and slicing's approach to finding faults. Program slicing is a fine-grained analysis at the statement level that can be used to inspect a failing program to help locate the cause of the failure. Our work focuses on failures that occur due to a specific edit between program versions, and our analysis is at the method level.

7.4 Techniques for Avoiding Recompilation

Existing techniques to avoid unnecessary recompilation use dependences between compilation units of a program to calculate which other units (i.e., clients) might require recompilation. This may be necessary, for example, if a specific compilation unit that defines functions or types is changed. This calculation uses interunit dependences that can be supplied by the programmer (i.e., as in the UNIX *make* [26]) or based on derived syntactic or semantic relationships. These dependences, describing clients of changed program constructs, are *incomparable* to the dependences used in *Crisp* that capture necessary additions to user-selected fine-grained changes required to form a minimal syntactically valid edit, because each captures different information.

Here, we briefly summarize several approaches to avoiding recompilation as representative of this research area. These techniques differ in their definitions of dependence and the granularity of the compilation units used, (i.e., files, classes or modules [27], [28]).

The earliest work was *smart recompilation* by Tichy et al. [29], [30], which defined dependences between compilation units, induced by *Pascal include* files that contained global constants and type definitions. Syntactic dependences were constructed between *include* files and those Pascal code files (i.e., $*.p$ files) which contained references to the include-defined constructs (e.g., types, constants). Tichy et al. later compared several smart recompilation approaches [30] empirically to quantify their benefits on several Ada programs, finding a savings of approximately 50 percent of the recompilation effort. Burke and Torczon [31] described semantic dependences between procedures derived from interprocedural data-flow information for Fortran programs. Their dependences were calculated using the alias, side-effect, reference, and constant-value information associated with each subroutine, assuming that this information might have been used to enable optimizations during compilation. Their technique was capable of fine-grained recompilation decisions on a procedure level. More recently, Dmitriev [32] used information provided in Java class files to calculate syntactic dependences between program constructs (e.g., fields and methods). His approach, called *smart dependency checking*, was to aggregate these dependences in order to ascertain the clients of a class (i.e., classes referencing members of another class). Thus, when the code for a class changes, its client classes are marked for recompilation. This automates the creation of dependences which can be used with *make* for Java programs.

8 Conclusions

This paper describes *Crisp*, a tool to automatically construct intermediate versions of a program that has been edited, in order to find those parts of the edit that have resulted in a failing regression test. *Crisp* uses the results from *Chianti*, a tool for semantic change impact analysis that decomposes an edit into atomic changes and finds those parts of the edit that affect a particular test. Given a failing test, *Crisp* displays its affecting changes (and their prerequisites), allowing a programmer to select those changes to be applied to the original program, in order to create an intermediate version. Then, *Crisp* automatically constructs a valid Java program from the selected set of changes and allows the programmer to execute the test on this intermediate version, in order to determine if any of her selected changes are failure-inducing.

We have presented initial experiences with using *Crisp* in case studies on nine version pairs of two moderate-sized Java programs, *Daikon* and the *Eclipse jdt compiler*. In these studies, although we were unfamiliar with these programs, we succeeded in finding the failure-inducing changes in 67 of the 68 failing tests! In most cases, the initial focus on possible failure-inducing changes by *Chianti* reduced the set of all changes in the edit by an order of magnitude to those changes possibly affecting a failing test. Then, exploration with *Crisp* enabled us to identify one to four changes that were failure-inducing for each test, (another order-of-magnitude reduction). These findings are promising in that they show that *Crisp* can potentially assist programmers in focusing on a very small subset of changes that has altered the behavior of a regression suite. This narrowing of programmer focus is invaluable, especially when a substantial edit has occurred. Our case studies also revealed some limitations of our current implementation of *Crisp*, which we are working on fixing.

We also presented extensive discussions of the change dependences that exist between different parts of an edit; these dependences enable *Crisp* to build valid Java programs by including all elements necessary for inclusion of programmer-selected changes. To our knowledge, this is the first such classification of edits of elements of an object-oriented program. We believe these dependences may help in program refactoring, which we will investigate in our future work.

We also are working on *JUnitCIA* [33], a tool built on *JUnit* and *Chianti* (in Eclipse) that classifies atomic changes with respect to the tests they affect in order to identify likely sources of test failure. In the future, we plan to integrate *Crisp*, *JUnitCIA*, and *Chianti* to automate the entire process of finding failure-inducing changes sets without the need for programmer intervention.

Acknowledgments

This research was supported by US National Science Foundation grant CCR-0204410 and in part by REU supplement CCR-0331797. The authors are very grateful for the contributions of theirr colleagues Frank Tip of IBM Research and Maximilian Stoerzer of the University of Passau to the authors' semantic change impact analysis research project.

References

[1] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley, "Chianti: A Tool for Change Impact Analysis of Java Programs," *Proc. ACM SIGPLAN Conf. Object Oriented Programming Languages and Systems (OOPSLA '04),* pp. 432-448, Oct. 2004.

[2] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Trans. Software Eng.,* vol. 27, no. 2, pp. 1-25, Feb. 2001.

[3] "The Eclipse IDE," http://www.eclipse.org/, 2006.

[4] B.G. Ryder and F. Tip, "Change Impact for Object Oriented Programs," *Proc. ACM SIGPLAN/SIGSOFT Workshop Program Analysis and Software Testing (PASTE '01),* June 2001.

[5] O. Chesley, X. Ren, and B.G. Ryder, "Crisp: A Debugging Tool for Java Programs," *Proc. Int'l Conf. Software Maintentance,* pp. 401-410, Sept. 2005.

[6] "Junit, Testing Resources for Extreme Programming," http://www.junit.org/, 2006.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification,* second ed. Addison-Wesley, 2000.

[8] X. Ren, F. Shah, F. Tip, B.G. Ryder, O. Chesley, and J. Dolby, "Chianti: A Prototype Change Impact Analysis Tool for Java," Technical Report DCS-TR-533, Rutgers Univ., Dept. of Computer Science, Sept. 2003.

[9] S.A. Bohner and R.S. Arnold, "An Introduction to Software Change Impact Analysis," *Software Change Impact Analysis,* S.A. Bohner and R.S. Arnold, eds., pp. 1-26, IEEE CS Press, 1996.

[10] M. Lee, A.J. Offutt, and R.T. Alexander, "Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software," *Proc. 34th Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS USA '00),* 2000.

[11] D.C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change Impact Identification in Object Oriented Software Maintenance," *Proc. Int'l Conf. Software Maintenance,* pp. 202-211, 1994.

[12] P. Tonella, "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis," *IEEE Trans. Software Eng.,* vol. 29, no. 6, pp. 495-509, 2003.

[13] J. Law and G. Rothermel, "Whole Program Path-Based Dynamic Impact Analysis," *Proc. Int'l Conf. Software Eng.,* pp. 308-318, 2003.

[14] A. Orso, T. Apiwattanapong, and M.J. Harrold, "Leveraging Field Data for Impact Analysis and Regression Testing," *Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng. (ESEC/FSE '03),* Sept. 2003.

[15] F. Tip, "A Survey of Program Slicing Techniques," *J. Programming Languages,* vol. 3, no. 3, pp. 121-189, 1995.

[16] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M.J. Harrold, "An Empirical Comparison of Dynamic Impact Analysis Algorithms," *Proc. Int'l Conf. Software Eng. (ICSE'04),* pp. 491-500, 2004.

[17] V. Rajlich and P. Gosavi, "Incremental Change in Object-Oriented Programming," *IEEE Software,* pp. 2-9, July/Aug. 2004.

[18] S. Gwizdala, Y. Jiang, and V. Rajlich, "Jtracker—A Tool for Change Propagation in Java," *Proc. Seventh European Conf. Software Maintenance and Reeng. (CVMR '03),* pp. 223-229, 2003.

[19] J. Buckner, J. Buchta, P. Maksym, and V. Rajlich, "Jripples: A Tool for Program Comprehension during Incremental Change," *Proc. 13th IEEE Int'l Workshop Program Comprehension (IWPC '05),* pp. 149-152, May 2005.

[20] J. Gustavsson, "A Classification of Unanticipated Runtime Software Changes in Java," *Proc. 19th IEEE Int'l Conf. Software Maintenance (ICSM '03),* pp. 4-12, 2003.

[21] A. Zeller, "Yesterday My Program Worked. Today, It Does Not. Why?" *Proc. Seventh European Software Eng. Conf./Seventh ACM SIGSOFT Symp. Foundations of Software Eng. (ESEC/FSE '99),* pp. 253-267, 1999.

[22] J. Lyle and M. Weiser, "Automatic Bug Location by Program Slicing," *Proc. Second Int'l Conf. Computers and Applications,* pp. 877-883, 1987.

[23] R.A. DeMillo, H. Pan, and E.H. Spafford, "Critical Slicing for Software Fault Localization," *Proc. 1996 ACM SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA '96),* pp. 121-134, 1996.

[24] P. Bunus and P. Fritzson, "Semi-Automatic Fault Localization and Behavior Verification for Physical System Simulation Models," *Proc. 18th IEEE Int'l Conf. Automated Software Eng.,* pp. 253-258, Oct. 2003.

[25] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating Faulty Code Using Failure-Inducing Chops," *Proc. Int'l Conf. Automated Software Eng.,* pp. 263-272, 2005.

[26] S.I. Feldman, "Make—A Program for Maintaining Computer Programs," *Software—Practice and Experience,* vol. 9, no. 4, pp. 255-65, 1979.

[27] C. Chambers, J. Dean, and D. Grove, "A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies," *Proc. 17th Int' Conf. Software Eng. (ICSE '95),* pp. 221-230, 1995.

[28] R. Hood, K. Kennedy, and H.A. Muller, "Efficient Recompilation of Module Interfaces in a Software Development Environment," *Proc. Second ACM SIGSOFT/SIGPLAN Software Eng. Symp. Practical Software Development Environments (SDE 2),* pp. 180-189, 1987.

[29] W.F. Tichy, "Smart Recompilation," *ACM Trans. Programming Languages and Systems,* vol. 8, no. 3, pp. 273-291, 1986.

[30] R. Adams, W.F. Tichy, and A. Weinert, "The Cost of Selective Recompilation and Environment Processing," *ACM Trans. Software Eng. Methodology,* vol. 3, no. 1, pp. 3-28, 1994.

[31] M. Burke and L. Torczon, "Interprocedural Optimization: Eliminating Unnecessary Recompilation," *ACM Trans. Programming Languages and Systems,* vol. 15, no. 3, pp. 367-399, 1993.

[32] M. Dmitriev, "Language-Specific Make Technology for the Java Programming Language," *Proc. 17th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02),* pp. 373-385, 2002.

[33] M. Stoerzer, B.G. Ryder, X. Ren, and F. Tip, "Find Failure-Inducing Changes Using Change Classification," Technical Report DCS-TR-05-582, Dept. of Computer Science, Rutgers Univ., Sept. 2005.

Xiaoxia Ren received the BS and MS degrees in computer science from Peking University, Beijing, China. She is currently a PhD candidate in computer science at Rutgers University, New Brunswick, New Jersey. Her research interests are in program analysis and its application in software engineering and program understanding tools.

Ophelia C. Chesley received the BS degree in decision science from Indiana University in Bloomington, Indiana. She is currently an MS candidate in computer science at Rutgers University. Her research interests include program understanding and software engineering.

Barbara G. Ryder is a professor of computer science at Rutgers University, New Brunswick, New Jersey. Dr. Ryder became a fellow of the ACM in 1998 and was selected as a CRA-W Distinguished Professor in 2004. She was selected for a Professor of the Year Award for Excellence in Teaching by the Computer Science Graduate Students Society of Rutgers University in 2003 and received the ACM SIGPLAN Distinguished Service Award in 2001. She was the General Chair of the 2003 Federated Conference on Research in Computing and served on the board of directors of the Computer Research Association (CRA) from 1998 to 2001. She was elected a member of the ACM council in 2000 and 2004 and served on the ACM SIGPLAN executive committee from 1989-1999 (as SIGPLAN chair, 1995-1997). She was a recipient of a US National Science Foundation Faculty Award for Women Scientists and Engineers (1991-1996). Dr. Ryder's research focuses on static and dynamic program analyses for object-oriented languages and practical software tools. Applications include change impact analysis, program understanding, software testing, and testing availability of Web services. She is a member of the IEEE Computer Society (http://www.cs.rutgers.edu/ryder).