Crisp: A Debugging Tool for Java Programs

Ophelia C. Chesley, Xiaoxia Ren and Barbara G. Ryder *

Division of Computer and Information Sciences
Rutgers University
ochesley, xren, ryder@cs.rutgers.edu

Abstract

Crisp is a tool (i.e., an Eclipse plug-in) for constructing intermediate versions of a Java program that is being edited in an IDE such as Eclipse. After a long editing session, a programmer usually will run regression tests to make sure she has not invalidated previously checked functionality. If a test fails unexpectedly, Crisp uses input from Chianti, a tool for semantic change impact analysis [8], to allow the programmer to select parts of the edit that affected the failing test and to add them to the original program, creating an intermediate version guaranteed to compile. Then the programmer can re-execute the test in order to locate the exact reasons for the failure by concentrating on those affecting changes that were applied. Using Crisp, a programmer can iteratively select, apply, and undo individual (or sets of) affecting changes and, thus effectively find a small set of failure-inducing changes.

1. Introduction

In regression testing, programmers develop a set of tests over time to exercise and confirm the fundamental characteristics of a piece of software. After a long code editing session, regression tests are executed to ensure that the changed code in the updated program version does not conflict with previous releases of the software. During this phase, any test cases that produce unexpected results indicate not only potential defects in the software, but that the edits introduced in the updated version interact negatively with previously tested versions. Since regression tests are written to test the overall behavior of the software, each test can be affected by changes in different Java classes and methods. When these tests fail, programmers are burdened with the task of searching through the program for the source(s) of the failure. Moreover, failure-inducing changes very often work together in a non-trivial manner.

In this paper, we describe Crisp, a tool to assist programmers in isolating relevant portions of an edit that directly cause the failure of regression tests. This work extends our earlier research prototype Chianti, an Eclipse plug-in that performs change impact analysis of Java programs[10, 8]. Chianti divides a program edit into its constituent parts known as atomic changes, identifies a set of regression tests that are impacted by these changes, and for each affected test, identifies the subset of changes (called affecting changes) that may affect its behavior. In essence, Chianti automatically gathers all the relevant changes pertaining to each regression test. Initial experiments with Chianti have evidenced promising results for these analyses [8].

While the subset of the edit that may affect a regression 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 for programmers. For large applications, parts of an edit are inter-related in many ways, and there can be more than one subset of changes that the programmer considers as failure-prone with respect to a test.

There are benefits in automating the iterative process of selecting changes of interest and applying them to the original software version to create intermediate source program versions. Each of these versions can then be tested using the regression test that failed earlier. Programmers can ignore certain changes that do not result in failure, and 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 the process, so that they do not need to be concerned with the syntactic inter-relationship of the changes, nor with manually changing any code until they have found the failure-inducing changes and are ready for focused debugging.

We have built Crisp, an Eclipse plug-in built on Chianti[8], that generates valid intermediate programs automatically for programmers, given their choice of the changes to include. We have conducted an initial case study using program versions from the Daikon system [5] that showed that Crisp can effectively assist programmers in locating the failure-inducing changes, even in an unfamiliar program. The contributions of this paper are:

- Elaboration of the dependence relationships among changes in an edit, allowing automatic grouping of atomic changes to build syntactically valid intermedi-
ate programs.
- Construction of the Crisp prototype as an Eclipse plug-in built on Chianti. Crisp is the bridge between change impact analysis and the ultimate goal of providing relevant information for programmers to identify the failure-inducing changes set.
- A case study using Daikon that shows how Crisp can help programmers narrow down the failure-inducing changes set.

The remainder of this paper is organized as follows. Section 2 introduces a running example that takes us through the change impact analysis process to locate the failure-inducing changes. In Section 3, we give a summary of the underlying change impact analysis, and discuss the dependences between atomic changes. Section 4 reports the engineering issues of building our tool. Section 5 discusses the change impact analysis process to locate the failure-inducing changes set. In Section 3, we give a summary of

2. Debugging Using Chianti and Crisp

We will use the example program of Figure 1 to illustrate our approach. The program of Figure 1(a) depicts a simple program comprising classes A, B, and C. Figure 1(b) shows an edited version of the program, where the changes are shown using underlining.

Associated with the program are three JUnit tests, Tests.test1, Tests.test2, and Tests.test3, which are shown in Figure 1(c). 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 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. Knowledge of change dependences allows us to localize bugs more quickly. If a set \( S \) of atomic changes is likely to expose a bug, then knowing that applying certain subsets of \( S \) will not lead to syntactically valid programs, focuses our attention on viable, interesting intermediate versions.

Figure 1(d) 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 \( A_2 \) is dependent on \( A_1 \). Consider, for example, the addition of the call to method inc() in method B.foo() (CM atomic change 4 in Figure 1(d)). 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.

The LC atomic change category models changes to the dynamic dispatch behavior of instance methods. In particular, an LC change \( \langle X.m(), Y \rangle \) models the fact that a call to method \( X.m() \) on an object of 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 program of Figure 1(a). 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. LC changes are also generated in situations where a dispatch relationship is added or removed as a result of a source code change affecting dynamic dispatch behavior. For example, addition of method C.bar() (i.e., AM atomic change 9) also results in LC atomic change 12.

Figure 1(e) shows the call graphs for the 3 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 \( \langle T,M \rangle \), 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

---

1 We present intuitive definitions of our change impact analysis that pertain to Crisp usage; a more formal treatment is available in [8].

2 Chianti can work with call graphs that have been constructed using static analysis or from the actual execution of the tests.
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);
    }
}

Figure 1. (a) Original version of example program. (b) Edited version of example program (underlining is used to show added code fragments. (c) Tests associated with the example of (a) and (b). (d) Atomic changes for the example program, with their inter-dependences. (e) Call graphs for the tests in the original program (a). (f) Call graphs for the tests in the edited program (b).

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 Figure 1(e), it is easy to see that (i) test1 is not affected and (ii) 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 Figure 1(f). 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: (i) all atomic changes for added methods (AM) and changed methods...
that correspond to a node in the call graph (in the edited program), (ii) atomic changes in the lookup change (LC) category that correspond to an edge in the call graph (in the edited program), and (iii) their transitively prerequisite atomic changes.

We can compute the affecting changes for test3 as follows. Observe, that the call graph for test3 in Figure 1(f) contains methods B.foo(), B.inc(), and C.bar(), and an edge labeled <C, A.bar()>. Node B.foo() corresponds to atomic change 4, which is dependent on atomic change 1, and node B.inc() corresponds to atomic change 2, which is dependent on atomic changes 1 and 3. Node C.bar() corresponds to atomic change 10, which is dependent on atomic changes 6 and 9. Finally, the edge labeled <C, A.bar() > corresponds to atomic change 11, which is also dependent on atomic change 9. Consequently, test3 is affected by atomic changes 1, 2, 3, 4, 6, 9, 10 and 11. There is no edge labeled <C, C.bar() >, so that atomic change 12 is not in the affecting changes set. Similarly, test2 is affected by atomic changes 1, 2, 3 and 4.

2.2. Debugging with Crisp.

The original program passed all the tests, but test3 failed in the edited version. As Figure 1(d) shows, there are 12 atomic changes for the entire program and 8 of them are considered affecting changes for test3. The question is: Which of those 8 changes are the likely reason(s) for the test failure? Our tool Crisp helps programmers locate the failure-inducing changes by allowing 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 automatically generates an intermediate program version by applying the selected atomic changes, as well as all the other necessary atomic changes needed to build a valid program.

For test3, a programmer may first guess that the change to method B.foo() is the reason for its failure. When she selects atomic change 4, Crisp automatically applies atomic change 1 prior to applying atomic change 4 to the original program. In our original atomic change definitions, AM represents adding a method declaration with an empty body. Crisp always adds the corresponding CM change for the same method to the intermediate version. (Section 4.1 discusses the granularity of editable changes in detail.) Since atomic changes 1 and 2 are considered inseparable, selecting atomic change 4 results in Crisp applying atomic changes 1, 2, and 3 prior to applying 4 to create the intermediate program P1. Figure 2(a) shows the intermediate program P1.

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 augments the necessary prerequisite changes and generates the intermediate program. Note that the intermediate program P1 is independent of the program’s development history, and the affecting changes are not necessarily from the same class or package.

The programmer can now execute test3 against P1 and find that it succeeds. She may then suspect that the newly added method C.bar() is the potential culprit in the edit. Having created P1 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 P2 shown in Figure 2(b) by applying atomic changes 9 and 10. Re-executing test3 on P2 results in a failure, revealing the set of failure-inducing affecting changes to be {6, 9, 10, 11}. With the help of Chianti and Crisp, programmers can effectively pinpoint the 4 failure-inducing changes out of 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 pair-wise comparison on their abstract syntax trees and decomposes the source code modifications into a set of inter-dependent atomic changes A, whose granularity is roughly at method level. Chianti handles the full Java programming language. 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), adding, changing and deleting a static initializer of a class (ASI/CSI/DSI).

Atomic changes have syntactic inter-dependences 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 $C_2$. A key aspect of our tool is the ability to automatically
class A {
    public int i, s;
    public void foo() { }
    public void bar() { s = i+1; }
    public int x;
}
class B extends A {
    public void foo() { s = i; }
    private void inc() { x++; }
}
class C extends B {
    public void foo() { x = i+1; inc(); }
}

(a) \( P_1 \) (b) \( P_2 \)

Figure 2. (a) Intermediate program after selecting atomic change 2 to the original version of example program. (b) Intermediate program after applying atomic changes 9, 10 to the original version of example program.

We use the following rules to determine the requirements of program fragments referenced in an atomic change.

**Rule 3.1 (Frame-Body Rule)**

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.

In our definition, all the adding changes (AC, AF, AM, A1 and ASI) and deleting changes (DC, DF, DM, DI and DSI) represent adding or deleting an empty element\(^4\). For example, a definition of a new class is decomposed as a set of atomic changes: adding the class definition, adding its members, and then changing the body of each of the member. The dependence between atomic change 1 and atomic change 2 in Figure 1 (d) is a trivial example of this rule. If we want to add a class \( C \) with initialized field \( f \), then \( AC(C) \prec AF(C.f) \prec CFI(C.f) \). Symmetrically, deleting a class requires deleting all the methods, fields, initializers and member classes defined in that class. In summary, this rule defines the dependences which involve how new Java constructs are added or deleted in the program.

A special case of the Frame-Body rule concerns anonymous classes and local inner classes which are usually defined inside a block (anonymous classes can be defined in the initializer of a field). According to the Frame-Body rule, we have to declare the enclosing element before defining the anonymous class or local inner class. For example, if we add a new method \( C.\text{foo}() \) and define a local class \( LocalC \) inside its body, then there exists a dependence \( AM(C.\text{foo}()) \prec AC(CS\text{LocalC}) \). We also define a dependence \( CM(C.\text{foo}()) \prec AC(CS\text{LocalC}) \), which is required by \textit{Crisp} to generate the intermediate programs. Section 4.1 discusses the detail about how \textit{Crisp} deals with changes to anonymous classes and local inner classes.

**Rule 3.2 (Define-Use Rule)**

A program element must be declared before any other program element can have a reference to it. Similarly, a program element can only be deleted when there is no other program element referring to it.

The Define-use rule captures the dependences incurred by the interactive usage of Java elements. In Figure 1 (d), the dependences between pairs of atomic changes (1, 4), (3, 2), (5, 8), (6, 8) all are generated from this rule. Other examples of this rule include: \( AC(A) \prec AC(B) \), if type \( B \) is a subtype of type \( A \), \( AC(A) \prec AM(X.\text{foo}()) \) if type \( A \) is used as the return type or any parameter type of method \( X.\text{foo}() \). Symmetrically, deleting a definition of a class, field, or method is dependent on deleting all the uses of this class, field or method.

Some Define-Use dependences are implicit. 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 \textit{Chianti} also reports a CM for each of class \( C \)'s explicitly defined constructors or reports a CM for the implicitly declared default method \( C.\text{init}() \). The changes to initializers or field initializers become prerequisites of the corresponding CM changes. Similarly, if changes are made to static initialization blocks (CSI) or class variables (CSFI) of class \( C \), then atomic change \( CM(C.\text{etinit}()) \) is reported and it is dependent on the CSI or CSFI change.

\(^4\) In this paper, AC and DC both represent changes to classes and interfaces.
Rule 3.3 (Abstract-Method Rule)
An abstract method must be implemented in all the sub-
classes of an abstract class, before its declaration in this
class. Similarly, the declaration of an abstract method must
be deleted before the deletion of its implementation in the
subclasses.

The Abstract-Method rule captures the way a new ab-
stract method is declared or deleted. For example, program
P defines an interface 2 with two subtypes: class A and B.
In the edited program P', we add a new declaration of ab-
stract method 2foo() into interface 2, and class A and class
B both provide the implementation of this method 2foo().
Chianti will report that AM(A.foo()) ≺ AM(I.foo())
and AM(B.foo()) ≺ AM(I.foo()); otherwise, applying
only atomic change AM(I.foo()) to program P will result
in an intermediate program P'' which will not compile.

Rule 3.4 (Dynamic-Dispatch Rule)
An LC change is dependent on the corresponding source
code changes that result in the dynamic dispatch change.

The LC atomic change category models changes to the
dynamic dispatch behavior of instance methods. In partic-
ular, LC(Y, X.m()) models the fact that a call to method
X.m() on an object of runtime type Y results in the selec-
tion of a different method in the edited program. LC
changes can be caused by edits that alter inheritance rela-
tions.

Addition or deletion of a class may result in LC changes.
Consider the example in Figure 1 (a), 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 run-
time type D, while in the edited program P', programmers
can invoke a call to method A.foo() on an object of type
D. Chianti will report that AC(D) ≺ LC(D, A.foo())5.

Addition or deletion of an overriding method may also
result in LC changes. Consider the example in Figure 1
(b), a new overriding method bar() is added to class C
in the edited program P'. Two LC changes are reported
in Figure 1 (d): LC(C, A.bar()) and LC(C, C.bar()), and
both are dependent of atomic change 9 (AM(C.bar())).

Changing the modifier of a class or changing the access
control of a method, for example, making an abstract class
C non-abstract, will result in LC changes. In the original
dynamic dispatch map, there is no entry with C as the run-
time receiver type, but the new dispatch map will contain
such an entry. Similarly, changing the access control of
a method could also result in 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, so this CM is a prerequisite of the new LC change.

4. Constructing Intermediate Versions

Regression test suites are usually executed before com-
mitting a new version of an application to a version control
repository. Depending on the maturity of an application,
source code changes between versions can comprise thou-
 sands of atomic changes. Similarly, a regression test suite
can contain hundreds of tests which exercise different meth-
ods with various inputs. 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 gen-
erated by Chianti for each affected test are the first step in
the process.

The affecting changes of a failed affected test can be ap-
plied incrementally to the original program to obtain an in-
termediate source program P1. A programmer can then re-
execute the affected test against P1 and, based on the out-
come of the test, isolate the few changes that potentially
cause the failure of the test. The process is iterative, yet te-
dious if done manually, as there can be many different ver-
sions of P1 depending on the original number of affecting
changes (and their combinations) that are of interest. We
therefore developed Crisp as an add-on tool to Chianti that
creates intermediate versions based on programmer selec-
tions.

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 two major tasks that Crisp
performs are (i) to gather and order all the prerequisites of
the affecting changes and present them in a dependence tree
format and (ii) to respond to a programmer’s selection of
an affecting change and automatically update the original
source program P with the affecting change as well as all of
its (indirect and direct) prerequisites so that a syntactically
correct program P1 results. The Eclipse Plug-in Develop-
ment 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 loca-
tions of every Java construct 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 as-
pects of Crisp that differ from the original usage of atomic
changes in Chianti.

4.1. Editable Changes

The granularity of the output of Chianti is at the atomic
change level; however, certain atomic changes by them-

5 Chianti also reports LC changes for other inherited methods from
library classes (e.g., LC(D, java.lang.Object.toString())), and de-
clares AC(D) as prerequisite to these LC changes.
selves do not constitute legitimate program edits. For example, *Chianti* distinguishes between an AM change and a CM change. An AM could reflect a change of visibility of a method such as from abstract to non-abstract. But the more common scenario of adding a method requires adding its declaration (AM) as well as the implementation of its body (CM). In this context, presenting the AM and the CM separately for adding a method public void foo() is confusing to the users. 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 re-execution of empty methods. Similar circumstances apply to other atomic changes as well. A user 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).

Given the potential use of *Crisp* as a testing and debugging tool, we therefore aggregate the most common atomic changes into *editable changes* based on their semantics as follows: (i) add/changeMethod – add the method declaration and body; (ii) change/deleteMethod – delete the method body and declaration; (iii) add/changeFieldInitializer – add a field variable, its type, and its initial value; and (iv) change/deleteFieldInitializer – delete a field variable, its type, and its initial value.

With editable changes, users now will be able to select a change of interest, for example without understanding the technical difference between an AM versus a CM when adding a new method to the original program. For an editable change (e.g., add/changeMethod), *Crisp* combines the prerequisites of the individual atomic changes and process them as a single set.

**Initializers.** Similarly to above, we initially intended to create editable changes such as add/changeInitializer that conform more naturally to the way users edit their programs. When a single class contains multiple initializers, however, we encountered difficulties.

```
class A {
    int i, month;
    { i = 10; } (1)  
    { months=12; } (1)
    { months=12; } (2)  
}

(a)          (b)
```

Figure 3. (a) Original program with 2 initializer blocks (1) and (2). (b) Edited program with 1 initializer block (1) - block (1) in original program has been deleted.

Consider the example of deleting an initializer block in Figure 3. Within the Eclipse abstract syntax tree representation, initializer blocks are numbered instead of named. Identifying {i = 10;} as block (1) and {months=12;} as block (2) in the original program would suffice as long as the number for these blocks uniquely identifies the block in the two program versions. However, the initializer blocks are numbered by their relative position in the class file. If you insert a new block before an existing one, new block becomes (1) and the original block becomes (2).

Therefore, in Figure 3, deleting the first initializer block results in the following atomic changes from *Chianti*: CI(A,2), DI(A,2), and CI(A,1). When *Chianti* compares the original abstract syntax tree with the edited version, initializer block (2) appears to be deleted and initializer block (1) appears to have changed, when in fact initializer block (1) has been deleted.

Originally in *Crisp*, CI and DI for block 2 were taken as a pair of atomic changes, hence selecting either one of them resulted in the deletion of the second initializer block from the original program. Selecting CI for block 1 by itself produces two identical initializer blocks of {months=12} in the same class. Thus initializer changes cannot be presented to users as separate edit units. Consequently, we have to treat initializers and static initializers differently from the rest of the editable changes. All atomic changes related to any initializers and static initializers of the same class are collectively processed in *Crisp* as one editable change. Given that initializers and static initializers constitute only a small fraction of atomic changes, in our experience, aggregating them should have minimal impact to the effectiveness of *Crisp* as a tool to generate the intermediate versions of a program.

**Local and Anonymous Classes.** The notion of combining atomic changes and presenting them to the users as editable changes 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. Nevertheless, from the perspective of an editable change, breaking a method body into statements that are contained within the local and anonymous classes versus those statements within the method but outside such classes, violates the practical focus of *Chianti* on method-level changes. We therefore have provided editable changes in *Crisp* only at the method level. When a local or an anonymous class is selected by a programmer, *Crisp* applies all the changes in the edit that affect the method body. This is further guaranteed by the Frame-Body Rule we discussed in section 3.1. In fact, any local/anonymous classes chosen by the programmer in *Crisp* are bound by the special case of Frame-Body Rule such that the change
dependence tree, regression test suite, that must contain some failing tests. In order to simulate activities during the software development life cycle well, we must have access to both the software source code and a comprehensive set of test cases. For our research group, in order to simulate activities during the software 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, from Daikon’s test suite that executed successfully in the November 11\(^{th}\) version, but failed in the edited version dated November 19\(^{th}\). We therefore used Chianti to generate the atomic changes for these two versions, confirm that the two tests are in fact affected, and calculate their affecting changes. The results were then passed to Crisp to create the editable changes and present them in the dependence tree window.

5. Case Studies using Daikon

It is a challenge to identify appropriate test data for Crisp obtainable from a real-world software project outside of our research group. In order to simulate activities during the software development life cycle well, we must have access to both the software source code and a comprehensive test suite, that must contain some failing tests. Since bugs in small programs can be found using traditional tools such as the GUI debugger in Eclipse, our techniques are applicable to software of at least moderate size.

In previous research, we had extracted 52 weeks of Daikon source code from a CVS repository for testing the effectiveness of Chianti (all year 2002 check-ins). In addition, we had observed that the number of test cases in the test suite grew from 42 to 62. We were not surprised to find that executing these test cases did not result in any failures as commonly, failed tests delay code check-in until the program was complete. 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, from Daikon’s test suite that executed successfully in the November 11\(^{th}\) version, but failed in the edited version dated November 19\(^{th}\). We therefore used Chianti to generate the atomic changes for these two versions, confirm that the two tests are in fact affected, and calculate their affecting changes. The results were then passed to Crisp to create the editable changes 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.

For the test testXor, there are 35 affecting changes; for testMinus, 34. These affecting changes are selected by Chianti from a total of 6093 atomic changes comprising the edit between the version pair from November 11\(^{th}\) and 19\(^{th}\).

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

Selecting editable changes without prerequisites turned out to be extremely effective. For test testMinus, we were able to locate an affecting change \( CM(MinusVisitor.shouldAdd()) \) whose applying to the original program caused failure. For testXor, selecting changes without prerequisites resulted in finding \( CM(XorVisitor.shouldAddInv2()) \), which caused failure.

In order to confirm our results, we applied all the editable changes except \( CM(MinusVisitor.shouldAdd()) \) to the original program and re-executed testMinus, which then succeeded. This showed that the changed

![Figure 4. Screenshot from Crisp.](image)
method $CM(MinusVisitor.shouldAdd())$ was the only failure-inducing change for testMinus. However, for test testXor, the application of the complementary changes also resulted in test failure. We then continued our approach, selected other editable changes without prerequisites, and found $CM(XorVisitor.shouldAddInv1())$ as another failure-inducing change for test testXor. Neither atomic change $CM(XorVisitor.shouldAddInv1())$ nor $CM(XorVisitor.shouldAddInv2())$ 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.

From this experiment, we demonstrated the potential use of Crisp in assisting programmers to explore and locate failure-inducing changes for a regression test. Crisp calculates the set of affecting changes for each test to be about 0.5% of the original number of changes between November 11th and 19th. A programmer can narrow her focus further using Crisp and apply only a small subset of suspicious changes to the original program. In our experiments, the sizes of the subsets are 1 and 2 respectively. On the other hand, for coarser 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 intermediate program version is guaranteed to compile, based on Chianti’s dependence calculation. The ability to roll back the changes allows a programmer to explore in free form - changes can be aggregated in different ways and each exploration can be performed independently.

6. Related Work

Our Chianti prototype has already been described in [10, 8, 9]. 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 [8, 9], but they are not grouped into categories nor explained in the detail given here. There is no previous mention of Crisp in these papers.

Other areas of research relevant for comparison with Crisp are delta debugging and techniques for avoiding recompilation. Change impact analysis and regression testing are related to the analyses in Chianti (used by Crisp) and have been discussed extensively in [8]; space limits preclude us from considering them here.

6.1. Delta Debugging

In the work by Zeller et al. on delta debugging, the reason for a program failure is identified as a set of differences between program versions [13] 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 [13].

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 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. Second, delta debugging builds the intermediate versions by only using the structural differences between succeeding and failing program executions (e.g., changing one line or one character to generate an intermediate program version). Our model of dependences between atomic changes ensures that Crisp only builds meaningful intermediate versions of the program, 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 semi-automatic, 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.

6.2. 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 inter-unit dependences that can be supplied by the programmer (i.e., as in the UNIX make [6]) 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 summarize briefly 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 [3, 7]).

The earliest work was smart recompilation by Tichy [12, 1] 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 [1] empirically to quantify their benefits on several Ada programs, finding a savings of approximately 50% of the recompilation effort. Burke and Torczon[2] described semantic dependences between procedures derived from interprocedural dataflow 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 [4] used information provided in Java class files to calculate syntactic dependences between program constructs (e.g., fields, 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.

7. Conclusions

We have described our tool, Crisp, which can be used by a programmer to create intermediate versions of an edited program in order to reveal a set of changes responsible for the failure of a regression test. In our initial experience of using Crisp in two case studies on a moderate-sized program, we identified in one case 1 and in the other case 2 atomic changes that caused the failure of each of two regression tests after an edit. These changes were identified out of the 34/35 affecting changes for these two tests calculated by Chianti from a total of 6093 atomic changes in the edit. These findings are promising in showing 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.

We also are working on JUnitCIA [11], 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 programmer intervention.

References