CHANGE IMPACT ANALYSIS FOR JAVA PROGRAMS AND APPLICATIONS

BY XIAOXIA REN

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Barbara Gershon Ryder

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2007

ABSTRACT OF THE DISSERTATION

Change Impact Analysis for Java Programs and Applications

by Xiaoxia Ren

Dissertation Director: Barbara Gershon Ryder

Small changes can have major and nonlocal effects in object oriented languages, due to the extensive use of subtyping and dynamic dispatch. This makes it difficult to understand value flow through a program and complicates life for maintenance programmers. Change impact analysis provides feedback on the semantic impact of a set of program changes.

The change impact analysis method presented in this thesis presumes the existence of a suite of regression tests associated with a Java program and access to the *original* and *edited* versions of the code. The primary goal of our research is to provide programmers with tool support that can help them understand why a test is suddenly failing after a long editing session by isolating the changes responsible for the failure. The tool analyzes two versions of an application and decomposes their difference into a set of atomic changes. Change impact is then reported in terms of affected tests whose execution behavior may have been modified by the applied changes. For each affected test, it also determines a set of affecting changes that were responsible for the test's modified behavior.

The first contribution of this thesis is the demonstration of the utility of the basic

change impact analysis framework of [51], by implementing a proof-of-concept prototype, *Chianti*, and applying it to *Daikon*, for an experimental validation.

The second contribution is the definition and implementation of the dependences between atomic changes. Extensive experiments show that our dependences can help build the intermediate programs automatically in most cases.

Another contribution is the heuristics for ranking the atomic changes for fault localization. This thesis proposes a heuristic that ranks method changes that might have affected a failed test, indicating the likelihood that they may have contributed to a test failure. Our results indicate that when a failure is caused by a single method change, our heuristic ranked the failure-inducing change as number 1 or number 2 of all the method changes in 67% of the delegate tests (i.e., representatives of all failing tests).

Acknowledgements

I would like to thank my advisor, Professor Barbara Gershon Ryder, for her constant support, unconditional help, and her belief in my abilities. She was always my mentor and she taught me many things that influenced my professional and personal growth. I am very grateful to all the PROLANGS members, and the PROLANGS reading group members for providing interesting discussion and challenging and productive research environment. I am also thankful to Dr. Frank Tip and Dr. Maxillian Stoerzer for their help during my studies, and for many things I learned from them. I would like to thank Michael Ernst and his research group at MIT for the use of their data.

I am deeply thankful to my dearest husband, Chen Fu, for his constant encouragement, help and support, and for always having faith that I do a good job in my research area.

Dedication

This thesis is dedicated to my lovely daughters, Sophia and Arianna, for all the happiness they brought to me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Software Maintenance

Software evolves over time for many reasons: fixing bugs, enhancing to add new functionality and new features, refactoring to improve the performance and other attributes, adapting to new algorithms or to modified environments, etc. So changes to the software are inevitable even if the system was developed 'right first time'.

Software maintenance is the most costly and difficult phase in the software life cycle. Nowadays, software becomes larger and larger, with many components interacting with each other, which complicates the maintenance task, a small change may cause the entire system to fail. Furthermore, many legacy systems written 15-20 years ago are still in service, and they have undergone many generations of changes. The maintainers are rarely the original designer of the system and usually lack a complete understanding of design and specifications of the program. As software evolves, the task of maintaining it becomes more complex and more expensive. Software maintenance task has been estimated to be more than 50% of the total software life cycle cost [24].

1.2 Change Impact Analysis Overview

One difficulty in software maintenance is to understand the maintained software system. Object-oriented programming languages present many challenges for program understanding. The extensive use of subtyping and dynamic dispatch in object-oriented programming languages make it difficult to understand value flow through a program. Moreover, a seemingly small source code change can have unexpected and nonlocal effects, which means it may ripple throughout the system to have major unintended

impact elsewhere. For example, adding a method to an existing class may affect the dispatch behavior of virtual method calls throughout the program. Adding the creation of an object may affect the behavior of virtual method calls that are not lexically near the allocation site. This *non-locality of change impact* is qualitatively different and more important for object-oriented programs than for imperative programs. For example, in C programs a precise call graph can be derived from syntactic information alone, except for the typically few calls through function pointers [38]. As a result, maintenance programmers, who need to fix bugs or add enhancements to object-oriented systems are often hesitant to make invasive changes because of the unforeseen effects that these changes might have.

As a result, software developers need mechanisms to understand how a change to a software system may affect the rest of the system. This process is called *change impact analysis* [5, 35, 40, 51, 41, 43], which consists of a collection of techniques for determining the effects on the other components of the system of source code modifications.

Change impact analysis can help improving programmer productivity in several ways. The impact analysis information can be used for planning changes, doing trade-offs between changes, implementing changes, and tracing the effects of changes. If a programmer knows the potential effects of changes before the changes are implemented, it will be easy to perform changes more accurately and understand the consequences of proposed software changes during development and maintenance. If there are many candidate changes to satisfy the same changing requirement, the impact analysis information can be used to evaluate the cost of each change (e.g., the scope of the effects of the change, the possible security issues involved in the changes, and etc.) and allows a programmer to do trade-offs between alternative changes.

Change impact analysis can also be used for selective regression testing by determining the sets of tests that need to be rerun after changes are made. Regression testing is a software maintenance activity. It refers to the repetition of tests after changes to confirm the fundamental functionalities of a program is unchanged except insofar as required by the change. To save time and effort, only those tests that are affected by the changes need to be rerun. Rerunning too many tests in the system will increase

the cost of testing, but rerunning too few tests in the system might adversely affect the quality of the software. Change impact analysis can determine the set of tests affected by the changes and thus needed to rerun.

Previous approaches for dynamic change impact analyses [35, 40, 41] are primarily concerned with the problem of *determining a subset of the methods in a program that were affected by a given set of changes*. That is, they first do a comparison of high-level program representations such as control flow graphs (e.g., see [50]) or Java InterClass Graphs [30], identifying the changes between two program versions, then find all or some of the constructs of the program that are potentially affected by the code changes. In summary, the change impact is evaluated by the impacted constructs of the source code changes.

1.3 Our Change Impact Analysis

In contrast, our technique is based on the framework proposed in [51], which is concerned with the problem of *isolating a subset of the changes that affect a given test*. The change impact analysis method presented in this thesis presumes the existence of a suite $\mathcal{T}$ of regression tests[1] associated with a Java program, and access to the *original* and *edited* versions of the code. Our analysis comprises the following steps:

1. A source code edit is analyzed to obtain a set of *inter-dependent* atomic changes $\mathcal{A}$, whose granularity is (roughly) at the method level. These atomic changes include all possible effects of the edit on dynamic dispatch.

2. A call graph is constructed for each test in $\mathcal{T}$. Our method can use either dynamic call graphs that have been obtained by tracing the execution of the tests, or static call graphs that have been constructed by a static analysis engine.[2]

3. For a given set $\mathcal{T}$ of regression tests, the analysis determines a subset $\mathcal{T}'$ of $\mathcal{T}$ that is *potentially affected* by the changes in $\mathcal{A}$, by correlating the changes in $\mathcal{A}$

[1] In the rest of this thesis, we will use the term "regression test" to refer both to unit tests often used for validation after software changes and other sorts of regression tests.

[2] For all the data shown in the thesis, we use dynamic call graphs.

against the call graphs for the tests in $\mathcal{T}$ in the *original* version of the program.

4. For a given test $t_i \in \mathcal{T}'$, the analysis can determine a subset $\mathcal{A}'$ of $\mathcal{A}$ that contains all the changes that may have affected the behavior of t_i. This is accomplished by constructing a call graph for t_i in the *edited* version of the program, and correlating that call graph with the changes in $\mathcal{A}$.

As mentioned before, all of the impact analysis techniques previous to ours focus on finding *constructs of the program potentially affected by code changes*. Our change impact analysis aims to find *a subset of the changes that impact a test whose behavior has (potentially) changed*. Our research can improve productivity by:

- reducing the amount of time and effort needed in running regression tests, by determining that some tests are guaranteed not to be affected by a given set of changes, and
- reducing the amount of time and effort spent in debugging, by determining a safe approximation of the changes responsible for a given test's failure [51, 49, 48], and
- allowing programmers to experiment with different edits, observe the code fragments that they affect, and use this information to determine which edit to select and/or how to augment test suites [12, 45].

The primary goal of our research is to provide programmers with tool support that can help them understand why a test is suddenly failing after a long editing session by 1) isolating the atomic changes responsible for the failing test; 2) ranking the atomic changes that might have affected the failing test, indicating the likelihood that they may have contributed to the test failure; and 3) facilitating the automatic construction of the intermediate programs from user-selected atomic changes, further narrowing down the failure causes by rerunning the failed tests on the intermediate programs.

1.4 Contributions

The work presented in this thesis makes the following contributions.

1.4.1 *Chianti*–the Prototype of Change Impact Analysis

The first contribution of this thesis is a demonstration of the utility of the basic change impact analysis framework of [51], by implementing a proof-of-concept prototype, *Chianti*. *Chianti* greatly extends the originally specified techniques [51] to handle the entire Java language(J2SE 1.4), including such constructs as anonymous classes, initializers, and overloading. This entailed extending the model of atomic changes and their inter-dependences. In addition, we present experimental validation of the utility of change impact analysis by determining the percentages of affected tests and affecting changes for 40 versions of Daikon [21, 22] in 2002. For the 39 sets of changes between these versions, we found that, on average, 52% of the tests are potentially affected. Moreover, for each potentially affected test, on average, only 3.95% of the atomic changes affected it. This is a promising result with regard to the utility of our technique for enhancing program understanding and debugging. In addition, *Chianti* has been integrated closely with Eclipse [18], a widely used open-source development environment for Java, to make it possible for programmers to use and further extend the functionality of *Chianti* from Eclipse.

1.4.2 Dependences between Atomic Changes

The second contribution of this thesis is the definition and implementation of the notion of dependences between atomic changes.[3] Atomic changes have syntactic interdependences which induce a partial ordering $\prec$ on them, with transitive closure $\preceq^*$. $C_1 \prec C_2$ denotes that C_1 is a prerequisite for C_2. These dependences can be used to determine that certain changes are guaranteed *not* to affect a given test, and to construct syntactically valid intermediate versions of the program that contain some, but not all atomic changes. Three kinds of dependences are defined between atomic changes to ensure the compilability of the intermediate programs. *Structural dependences* capture the necessary sequences that occur when new Java elements are added or deleted in

[3]Dependences were not explicitly described in [51]

a program. *Declaration dependences* capture all the necessary Java element declarations that are required to create a valid intermediate version. *Mapping dependences* 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. *Crisp* [12, 13], built by Ophelia Chesley, relies on this automated computation of underlying inter-dependences between atomic changes to generate an intermediate program from user-specified atomic changes. Extensive experiments show that our dependences can help build the intermediate programs automatically in most cases [45, 11].

1.4.3 Heuristic Ranking of Edits

Another contribution of the thesis is a heuristic for ranking the atomic changes for fault localization. We propose a heuristic to rank method changes that might have affected a failing test, indicating the likelihood that they may have contributed to the test failure. The heuristic is based on the number of ancestors and descendants of a method in the test's call graph, as well as the calling relationships between changed methods. Our results indicate that when a failure is caused by a single method change, our heuristic ranked the failure-inducing change within the top 2 over all the method changes in 67%(8 over 12) of the *delegate* tests (i.e., representatives of all failing tests). Even when the failure is caused by some combination of the changes, rather than a single change, our heuristic still helps.

1.5 Thesis Outline

The rest of this thesis is organized as follows. The general framework of change impact analysis is presented in Chapter 2. The prototype tool of change impact analysis and its evaluation is presented in Chapter 3. Chapter 4 describes the syntactic dependences between atomic changes. Chapter 5 describes how these syntactic dependences are used to help automatic construction of intermediate programs from user-specified atomic changes. Chapter 6 presents a heuristic ranking of atomic changes in edits for fault localization. Related work is discussed in Chapter 7. Chapter 8 presents a summary of

the thesis and directions for future work.

The refinement and extension of the model of atomic changes, the prototype tool for change impact analysis and its evaluation were originally presented in [48, 47]. The data presented in the thesis are new collected, since we extended the model of atomic changes and re-engineered the tool. The definition of the syntactic dependences between atomic changes and its application to automatic construction of intermediate programs were originally presented in [12, 45, 13]. The syntactic dependences were extended in the thesis to accommodate the new model of atomic changes and allow the automatic construction of intermediate programs in more complicated cases. The heuristic ranking of atomic changes for fault localization was presented in [46].

Chapter 2

The Model of Change Impact Analysis for Java Programs

Regression tests are developed by programmers over time to confirm the fundamental functionality 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 non-trivial combinations of changes.

One of the primary goals of our research is to provide programmers with tool support that can help them understand why a test is suddenly failing after a long editing session by isolating the changes responsible for the failure.

2.1 Overview of Approach

This section gives an informal overview of the change impact analysis methodology originally presented in [51]. Our approach first determines, given two versions of a program and a set of tests that execute parts of the program, the *affected tests* whose behavior may have changed. Then, in a second step, for each test whose behavior was affected, a set of *affecting changes* is determined that may have given rise to that test's changed behavior. Our method is conservative in the sense that the computed set of affecting changes is guaranteed to contain at least every change that may have caused changes to the test's behavior.

We will use the example program of Figure 2.1(a) to illustrate our approach.[1] Figure 2.1(a) depicts two versions of a simple program comprising classes `A`, `B`, and `C`. The original version of the program consists of all the program text except for the 7 program fragments shown in boxes; the edited version of the program consists of all the program text including the program fragments shown in boxes. Associated with the program are 3 tests, `Tests.test1()`, `Tests.test2()`, and `Tests.test3()`.

2.1.1 Atomic Changes and Inter-dependences

Our change impact analysis relies on the computation of a set of *atomic changes* that capture all source code modifications at a semantic level that is amenable to analysis. We 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**).[2]

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 determine that certain changes are guaranteed *not* to affect a given test, and to construct syntactically valid intermediate versions of the program that contain some, but not all atomic changes. If a set S of atomic changes is known to expose a bug, then the knowledge that applying certain subsets of S does not lead to syntactically valid programs, can be used to locate bugs more quickly. More details about different categories of the syntactic dependences are discussed in Chapter 4.

Figure 2.1(b) shows the atomic changes that define the two versions of the example program, numbered 1–13 for convenience. Each atomic change is shown as a box, where the top half of the box shows the category of the atomic change (e.g., **CM** for changed

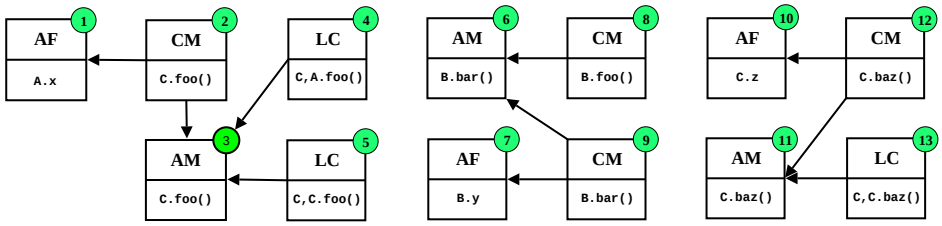
[1] This example was used in our previous paper [48]

[2] There are a few more categories of atomic changes that are not relevant for the example under consideration that will be presented in Section 2.3.

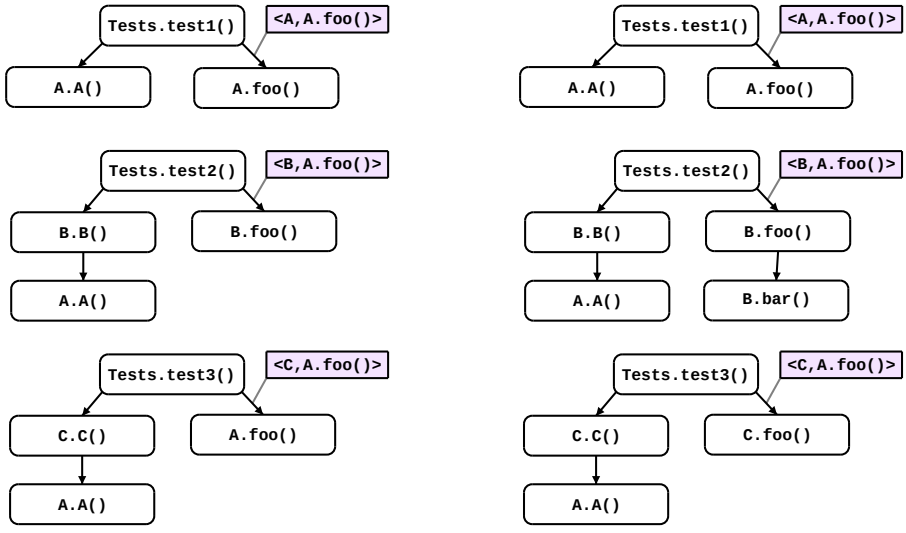
```
class A {
  public A(){ }
  public void foo(){ }
  public int x;
}
class B extends A {
  public B(){ }
  public void foo(){ B.bar(); }
  public static void bar(){ y = 17; }
  public static int y;
}
class C extends A {
  public C(){ }
  public void foo(){ x = 18; }
  public void baz(){ z = 19; }
  public int z;
}
```

```
class Tests {
  public static void test1(){
    A a = new A();
    a.foo();
  }
  public static void test2(){
    A a = new B();
    a.foo();
  }
  public static void test3(){
    A a = new C();
    a.foo();
  }
}
```

(a)



(b)



(c) **(d)**

Figure 2.1: **(a)** Example program with 3 tests. Added code fragments are shown in boxes. **(b)** Atomic changes for the example program, with their inter-dependences. **(c)** Call graphs for the tests before the changes were applied. **(d)** Call graphs for the tests after the changes were applied.

method), and the bottom half shows the method or field involved (for **LC** changes, both the class and method involved are shown). An arrow from an atomic change A_1 to an atomic change A_2 indicates that A_1 is dependent on A_2.

Consider, for example, the addition of the call `B.bar()` in method `B.foo()`. This source code change resulted in atomic change 8 in Figure 2.1(b). Observe that adding this call would lead to a syntactically invalid program unless method `B.bar()` is also added. Therefore, atomic change 8 is dependent on atomic change 6, which is an **AM** change for method `B.bar()`. The observant reader may have noticed that there is also a **CM** change for method `B.bar()` (atomic change 9). This is the case because our method for deriving atomic changes decomposes the source code change of adding method `B.bar()` into two steps: the addition of an empty method `B.bar()` (**AM** atomic change 6 in the figure), and the insertion of the body of method `B.bar()` (**CM** atomic change 9 in the figure), where the latter is dependent on the former. Observe that addition of `B.bar()`'s body requires that field `B.y` be added to class `B`. Hence, there is a dependence of atomic change 9 on **AF** atomic change 7, which represents the addition of field `B.y`. Notice that our model of dependences between atomic changes correctly captures the fact that adding the call to `B.bar()` to the body of `B.foo()` requires that a method `B.bar()` is added, but *not* that field `B.y` is added.

The **LC** atomic change category models changes to the dynamic dispatch behavior of instance methods. In particular, an **LC** change $(Y, X.m())$ models the fact that a call to method $X.m()$ on an object of type Y results in the selection of a different method. Consider, for example, the addition of method `C.foo()` to the program of Figure 2.1(a). As a result of this change, a call to `A.foo()` on an object of type `C` will dispatch to `C.foo()` in the edited program, whereas it used to dispatch to `A.foo()` in the original program. This change in dispatch behavior is captured by atomic change 4. **LC** changes are also generated in situations where a dispatch relationship is added or removed as a result of a source code change. For example, atomic changes 5 (defining the behavior of a call to `C.foo()` on an object of type `C`) and 13 (defining the behavior of a call to `C.baz()` on an object of type `C`) occur due to the addition of methods `C.foo()` and `C.baz()`, respectively.

2.1.2 Affected Tests

In order to identify those tests that are affected by a set of atomic changes, we have to construct a *call graph* for each test. The call graphs used in this thesis contain one node for each method, and edges between nodes to reflect calling relationships between methods. Our analysis can work with call graphs that have been constructed using static analysis, or with call graphs that have been obtained by observing the actual execution of the tests. In the experiments reported in this thesis, dynamic call graphs are used.

Figure 2.1(c) 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 $< T, M >$, where T is the run-time 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) either contains a node that corresponds to a changed method (**CM**) or deleted method (**DM**) change, or if its call graph contains an edge that corresponds to a lookup change (**LC**). Using the call graphs in Figure 2.1(c), it is easy to see that: (i) `test1` is not affected, (ii) `test2` is affected because its call graph contains a node for `B.foo()`, which corresponds to **CM** change 8, and (iii) `test3` is affected because its call graph contains an edge corresponding to a dispatch to method `A.foo()` on an object of type `C`, which corresponds to **LC** change 4.

2.1.3 Affecting Changes

In order to compute the changes that affect a given affected test, we need to construct a call graph for that test in the edited version of the program. These call graphs for the tests are shown in Figure 2.1(d).[3] The set of atomic changes that affect a given affected test includes: (i) all atomic changes for added methods (**AM**) and changed methods (**CM**) that correspond to a node in the call graph (in the edited program),

[3] The call graph for `test1` in the edited version of the program is not necessary for our analysis because `test1` was not affected by any of the changes, and is included in the figure solely for completeness.

(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.

As an example, we can compute the affecting changes for `test2` as follows. Observe, that the call graph for `test2` in the edited version of the program contains methods `B.foo()` and `B.bar()`. These nodes correspond to atomic changes 8 and 9 in Figure 2.1(b), respectively. Atomic change 8 requires atomic change 6, and atomic change 9 requires atomic changes 6 and 7. Therefore, the atomic changes affecting `test2` are 6, 7, 8, and 9. Informally, this means that we can automatically determine that `test2` is affected by the addition of field `B.y`, the addition of method `B.bar()`, and the change to method `B.foo()`, but *not on any of the other source code changes*! In other words, we can safely rule out 9 of the 13 atomic changes as the potential source for `test2`'s changed behavior.

To conclude our discussion of the example program of Figure 2.1, consider the atomic changes 10, 11, 12, and 13 corresponding to the addition of field `C.z` and method `C.baz()`, respectively. These atomic changes do not affect any of the tests, indicating that additional tests are needed.

2.2 Formal Definitions of Affected Test and Affecting Changes

Figure 2.2 shows a a modified version of the equations in first presented in [51].[4] We will use them to more formally define how we find affected tests and their corresponding affecting atomic changes, in general.

Assume the original program P is edited to yield program P', where both P and P' are syntactically correct and compilable. Associated with P is a set of tests $\mathcal{T} = t_1,...,t_n$. The call graph for test t_i on the original program, called G_{t_i}, is described by a subset of P's methods $Nodes(P, t_i)$ and a subset $Edges(P, t_i)$ of calling relationships between P's methods. Likewise, $Nodes(P', t_i)$ and $Edges(P', t_i)$ form the call graph G'_{t_i} on the edited program P'. Here, a calling relationship is represented as $D.n \rightarrow_{B,X.m} A.m$, indicating

[4] We change some equations to make them consistent with the representation of the call graph nodes and edges.

possible control flow from method $D.n$ to method $A.m$ due to a virtual call to method $X.m$ on an object of type B, where type B must be a subtype of type A, and A must be subtype of type X (i.e., $B{\leq^*}A{\leq^*}X$).[5] In these definitions, we implicitly make the usual assumptions [30], namely that execution of the program is deterministic and that the library code used and the execution environment (e.g., JVM, operating system, the contents of non-Java files and directories, etc.) itself remain unchanged.

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

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

Figure 2.2: Equations to obtain affected tests and affecting changes, where $B{\leq^*}A{\leq^*}X$.

$\mathit{AffectedTests}(\mathcal{T}, \mathcal{A})$ is a subset of $\mathcal{T}$ containing only those tests whose behavior may be affected by changes in $\mathcal{A}$. This comprises any test that traverses a changed method (**CM**) or deleted method (**DM**), as well as any test that contains a virtual dispatch whose behavior may have changed. $\mathit{AffectingChanges}(t, \mathcal{A})$ is a subset of the changes in $\mathcal{A}$ that may affect the behavior of a specific test t. This includes all atomic changes for added methods (**AM**) and changed methods (**CM**) that correspond to a node in the call graph (in the edited program), as well as any dynamic dispatch change that corresponds to an edge in the call graph (in the edited program), and all of their transitively prerequisite atomic changes.

Note that in $\mathit{AffectingChanges}(t, \mathcal{A})$ equation in Figure 2.2, we only focus on those **LC** changes whose runtime receiver type is different from the declaring type of the static method at call site (i.e., $\mathbf{LC}(B, X.m)(B{<^*}X)$, when comparing the dynamic dispatch

[5] $B{<}X$ means that type B is a direct descendant of type X; $B{\leq}X$ means that type B is a direct descendant of type X, or $B = X$; $B{<^*}X$ means that type B is a descendant of type X, and $B \neq X$; $B{\leq^*}X$ means that type B is a descendant of type X, or $B = X$.

changes with the calling edges in the call graph. The purpose of this restriction is to minimize the set of affecting changes presented to the programmer. That is, we ignore the change $\mathbf{LC}(B, X.m)$ with $B = X$ (and thus $B = A = X$ in the call graph). Because in this case, the call $D.n \rightarrow_{B,\ X.m} A.m$ becomes $D.n \rightarrow_{A,\ A.m} A.m$, the target $A.m$ must be a newly added method (**AM**) that results in the change $LC(A, A.m)$, which is already reported in the first part of the equations.

2.3 Atomic Changes

Our analysis assumes the existence of an original program P and a changed program P' derived from P. Both P and P are assumed to be syntactically correct and compilable. As previously mentioned, a key aspect of our analysis is the step of uniquely decomposing a source code edit into a set of inter-dependent *atomic changes*, as defined in Table 2.1. These have two important characteristics. First, their granularity matches our analysis; Second, any source code edit can be broken up into a *unique* set of atomic changes.

In the original formulation [51], several kinds of changes, (e.g., changes to access rights of classes, methods, and fields, addition/deletion of comments, and changes to the type hierarchy) were not modeled. Our analysis handles the full Java programming language (J2SE 1.4), which necessitated modeling several constructs not considered in the original framework [51], including abstract classes, interfaces, initializers,[6]nested classes[7] and visibility modifiers. Some of these constructs required the definition of new atomic changes; others were handled by augmenting the interpretation of atomic changes already defined. Table 2.1 lists the set of atomic changes, which includes the original 8 categories [51] plus 9 new defined atomic changes (marked with *).

[6] Instance initializers are blocks of executable code that may be used to initialize an instance when it is created.

[7] A nested class is any class whose declaration occurs within the body of another class or interface. Nested classes can be further classified into member classes, local classes and anonymous classes.

AF	Add a field
DF	Delete a field
***CFI**	Change definition of a instance field initializer
***CSFI**	Change definition of a static field initializer
***AI**	Add an empty instance initializer
***DI**	Delete an empty instance initializer
***CI**	Change definition of an instance initializer
***ASI**	Add an empty static initializer
***DSI**	Delete an empty static initializer
***CSI**	Change definition of an static initializer
AM	Add an empty method
DM	Delete an empty method
CM	Change body of a method
LC	Change virtual method lookup
AC	Add an empty class
DC	Delete an empty class
***CTD**	Change a type declaration

Table 2.1: Categories of atomic changes.

2.3.1 Field and Initializer Changes

AF and **DF** denote added and deleted fields respectively; similarly, **AI** and **DI** denote the set of added and deleted *instance* initializers respectively; and **ASI** and **DSI** denote the set of added and deleted *static* initializers, respectively. **CI** and **CSI** capture any change to an *instance* or *static* initializer, respectively. **CFI** and **CSFI** capture any change to an *instance* or *static* field, including (i) adding an initialization to a field, (ii) deleting an initialization of a field, (iii) making changes to the initialized value of a field, and (iv) making changes to a field modifier (e.g., changing a *static* field into a non-static field).

Changes to initializer blocks and field initializers also have repercussions for constructors or static initializer methods of a class. Specifically, if changes are made to initializers of instance fields or to instance initializer blocks of a class C, then there are two cases: (i) if constructors have been explicitly defined for class C, then our analysis will report a **CM** for each such constructor, (ii) otherwise, our analysis will report a

change to the implicitly declared method $C.\langle init \rangle()$ that is generated by the Java compiler to invoke the superclass's constructor without any arguments. Similarly, the class initializer $C.\langle clinit \rangle()$ is used to represent the method being changed when there are changes to a *static* field (i.e., **CSFI**) or *static* initializer (i.e., **CSI**).

2.3.2 Method Changes

AM and **DM** denote sets of added and deleted of methods declarations, respectively. Accommodating method access modifier changes from non-abstract to `abstract` or *vice versa*, and non-public to `public` or *vice versa*, required extension of the original definition of **CM** in [51]. In this thesis, **CM** comprises: (i) adding a body to a previously `abstract` method, (ii) removing the body of a non-abstract method and making it `abstract`, or (iii) making any number of statement-level changes inside a method body or any method declaration changes (e.g., changing the access modifier from `public` to `private`, adding a `synchronized` keyword or changing a `throws` clause).

Note that we decompose the source code change of adding a method `A.m()` into two steps: the addition of an empty method $\mathbf{AM}(A.m())$ and the insertion of the body of method $\mathbf{CM}(A.m())$, where the latter is dependent on the former.

2.3.3 Dynamic Dispatch Changes

LC represents changes in dynamic dispatch behavior that may be caused by various kinds of source code changes (e.g., by the addition of methods, by the addition or deletion of inheritance relations, or by changes to the access control modifiers of methods). **LC** is defined as a set of pairs $\langle Y, X.m() \rangle$, indicating that the dynamic dispatch behavior for a call to $X.m()$ on an object with run-time type Y has changed. **LC** changes can be classified as (i) newly added dynamic dispatch tuples (e.g., caused by declaring a new class/interface or method), (ii) deleted dynamic dispatch tuples (e.g., caused by deleting a class/interface or method), or (iii) dynamic dispatch tuples with changed targets (e.g., caused by adding/deleting a method or changing the access control of a class or method).

Changing a method's access modifier may result in changes to the dynamic dispatch

in the program (i.e., **LC** changes). For example, there is no entry for `private` or `static` methods in the dynamic dispatch map (because they are not dynamically dispatched), but if a `private` method is changed into a `public` method, then an entry will be added, generating an **LC** change that is dependent on the access control change, which is represented as a **CM**. Another example is that 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. Additions and deletions of import statements may also affect dynamic dispatch.

2.3.4 Class Changes

AC and **DC** denote added and deleted class/interface declarations, respectively. Programmers may also change the declaration of an existing class or interface, for example, moving a class in a hierarchy, or changing the visibility of a class. **CTD** represents any changes to the declaration of a class or an interface.[8] Considering the example program shown in Figure 2.3. In the original program, type `C` is declared as `class C extends B`, but in the edited program, the declaration changes to `class C extends A` and all the other type declarations remain the same. Our analysis reports **CTD**(C) to represent this type declaration change of class `C`. Although the whole subtree rooted at class `C` is moved, to keep the atomic changes clear and simple, our analysis won't report **CTD** changes for the subtypes of class `C`.

```
class A{ }                  class A{ }
class B{ }                  class B{ }
class C extends B{ }        class C extends A{ }
class D extends C{ }        class D extends C{ }
```

(a) **(b)**

Figure 2.3: Type Hierarchy change example

[8] Currently, type declaration changes to anonymous and local inner classes are not implemented since these changes occur very seldom.

2.3.5 Dependences

Atomic changes have inter-dependences which induce a partial ordering $\prec$ on a set of them, with transitive closure $\preceq^*$. Specifically, $C_1 \prec^* C_2$ denotes that C_1 is a prerequisite for C_2. This ordering determines a safe order in which atomic changes can be applied to program P to obtain a syntactically correct edited version P'' which, if we apply *all* the changes is P'. Consider that one cannot extend a class X that does not yet exist, by adding methods or fields to it (i.e., $\mathbf{AC}(X) \prec \mathbf{AM}(X.m())$ and $\mathbf{AC}(X) \prec \mathbf{AF}(X.f)$). These dependences are intuitive as they involve how new code is added or deleted in the program. Other dependences are more subtle. For example, if we add a new method `C.m()` and then add a call to `C.m()` in method `D.n()`, Our analysis will report a dependence $\mathbf{AM}(C.m()) \prec \mathbf{CM}(D.n())$. Figure 2.1(b) shows some examples of dependences among atomic changes. More details about dependences will be discussed in Chapter 4.

2.4 Special Issues

2.4.1 Overloading Methods

Overloading poses interesting issues for change impact analysis. Consider the introduction of an overloaded method as shown in Figure 2.4 in which 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 because the call of `R.foo(y)` in `R.bar()` resolves to `R.foo(Y)` instead of `R.foo(X)` [26] after the change, and affects the call graph of `Test.testBar()`. Therefore, our analysis reports a **CM** change for method `R.bar()` despite the fact that no textual changes occur within this method,[9] and creates a dependence: **AM***(R.foo(Y))* $\prec$ **CM***(R.bar())*.

To make our analysis safe in such cases, the algorithm for reporting such **CM** changes is conservative. For each **AM** or **DM**: $A.m(X1, ..., Xn)$, we first search for its

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

related overloaded methods in the whole hierarchy tree of type A. Then estimate all the possible callers of these overloaded methods,[10] and report **CM** for these callers.

```
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();
  }
}
```

Figure 2.4: Addition of an overloaded method. The added method is shown in a box.

2.4.2 Threads and Concurrency

Threads do not pose significant challenges for our analysis. The addition/deletion of `synchronized` blocks inside methods and the addition/deletion of `synchronized` modifiers on methods are both modeled as **CM** changes. Threads do not present significant issues for the construction of call graphs either, because the analysis discussed in this thesis does not require knowledge about the particular thread that executes a method. The only information that is required are the methods that have been executed and the calling relationships between them. If dynamic call graphs are used, as is the case in this thesis, this information can be captured by tracing the execution of the tests. If flow-insensitive static analysis is used for constructing call graphs [49], the only significant issue related to threads is to model the implicit calling relationship between `Thread.start()` and `Thread.run()`.

2.4.3 Exception Handling

Exception handling is not a significant issue in our analysis. Any addition or deletion or statement-level changes to a `try, catch` or `finally` block will be reported as a **CM** change. Similarly, changes to the `throws` clause in a method declaration are

[10] In implementation, we use *SearchEngine* in Eclipse to find all the callers of the overloaded methods.

also captured as **CM** changes. Possible inter-procedural control flow introduced by exception handling is expressed implicitly in the call graph; however, our change impact analysis correctly captures effects of these exception-related code changes. For example, if a method $f()$ calls a method $g()$, which in turn calls a method $h()$ and an exception of type E is thrown in $h()$ and caught in $g()$ before the edit, but in $f()$ after the edit, then there will be **CM** changes for both $g()$ and $f()$ representing the addition and deletion of the corresponding `catch` blocks. These **CM** changes will result in all tests that execute either $f()$ or $g()$ to be identified as affected. Therefore, all possible effects of this change are taken into account, even without the explicit representation of flow of control due to exceptions in our call graphs.

2.4.4 Anonymous Classes and Local Inner Classes

The growth of the Java language has introduced many new concepts. These new features bring additional facilities to programmers, but also introduce more difficulties for the analysis. One engineering problem encountered during implementation resulted from the absence of unique names for anonymous and local classes. In the Java virtual machine, anonymous classes are represented as `EnclosingClassName$`⟨`num`⟩, where the number assigned represents the lexical order of the anonymous class within its enclosing class. For local inner classes, the situation is very similar. In the Java virtual machine, local classes are represented as `EnclosingClassName$`⟨`num`⟩`$LocalClassName`. This naming strategy guarantees that all the class names in a Java program are unique.

However, when we compare and analyze two related Java programs, we need to establish a correspondence between classes and methods in each version to determine the set of atomic changes. The approach used is a *match-by-name* strategy in which two components in different programs match if they have the same name; however, when there are changes to anonymous or local inner classes, this strategy requires further consideration.

Figure 2.5 shows a simple program using anonymous classes with the code added by the edit shown inside a box. In this program, method `listJavaFiles(String)` lists all the Java files in a directory that is specified by its parameter. Anonymous

```
import java.io.*;
class Lister {
   static void listClassFiles(String dir){
     File f = new File(dir);
     String[] list = f.list(
       new FilenameFilter() { //anonymous class
         boolean accept(File f, String s){
           return s.endsWith(".class");
         }
       });
     for(int i = 0; i < list.length; i++)
       System.out.println(list[i]);
   }
    static void listJavaFiles(String dir){
      File f = new File(dir);
      String[] list = f.list(
      new FilenameFilter() { //anonymous class
        boolean accept(File f,String s){
            return s.endsWith(".java");
        }
      });
    for(int i = 0; i < list.length; i++)
      System.out.println(list[i]);
  }
}
```

Figure 2.5: Addition of an anonymous class. The added code fragments are shown inside a box.

class `Lister$1` implements interface `java.io.FilenameFilter` and is defined as part of a method call expression. Now, assume that the program is edited and a method `listClassFiles(String)` is added that lists all class files in a directory. This new method declares another similar anonymous class. Now, in the edited version of the program, the Java compiler will name this new anonymous class `Lister$1` and the previous anonymous class, formerly named `Lister$1`, will become `Lister$2`. Clearly, the *match-by-name* strategy cannot be based on compiler-generated names because the original anonymous class has different names before and after the edit.

To solve this problem, we use a new naming strategy that assigns each class a unique internal name. For top-level classes or member classes, the internal name is the same as the class name. For anonymous classes and local inner classes, the unique name consists of four parts: `enclosingClassName, enclosingElementName, selfSuperclassInterfacesName, sequenceNumber`. In this context: `enclosingClassName` is name of the nearest top level class or member class in which it is defined, `enclosingElementName` may be a method signature, a field name or an initializer number as appropriate for an anonymous or local class, `selfSuperclassInterfacesName` is a combination of the class name (only for a local class), superclass name and interface name, and `sequenceNumber` is used when more than one anonymous class is defined in the same code block and they all inherit from same type.

For the example in Figure 2.5, the unique internal name of the anonymous class in the original program is `Lister$listJavaFiles(String)$java.io.FilenameFilter$1`, while the unique internal name of the newly added anonymous class in the edited program is `Lister$listClassFiles(String)$java.io.FilenameFilter$1`. Similarly, the internal name of the original anonymous class in the edited program is `Lister$listJavaFiles(String)$java.io.FilenameFilter$1`. Notice that this original anonymous class whose compiler-generated names are `Lister$1` in the original program and `Lister$2` in the edited program, has the same unique internal name in both versions. With this new naming strategy, *match-by-name* can identify anonymous and local inner classes and report atomic changes involving

them.[11]

2.5 Limitations of the Model

Our current atomic changes model has some limitations for obtaining the affected tests, which means that in some special cases, our change impact analysis is *not safe* [50] in the sense that it does not guarantee that the set of affected tests contains at least every test whose behavior may have been affected.

2.5.1 Changes to Compile-time Constants

```
public interface I {
  int START = 2; // change to START = 1; in the edited program
  }
public class A implements I{
  public void foo(){
    int i = START;
    bar(i);
  }
  void bar(int i){
    switch(i){
      case 1:  throw new RuntimeException();
      case 2:  System.out.println(i);
        break;
    }
  }
}
public class Test extends TestCase {
  public void test1(){
    new A().foo();
  }
}
```

Figure 2.6: Changes to compile-time constants, our analysis fails to report affected test.

Consider the original example program in Figure 2.6. Suppose the programmer changes the value of the field `I.START` from `2` to `1` in the edited program. Originally, the test `Test.test1()` will print the value of `I.START`, but in the edited program, the test will throw a `RuntimeException()` and fail. However, our analysis fails to report

[11] This naming scheme can only fail when two anonymous classes occur within the same scope and extend the same superclass. If this occurs due to an edit, however, our analysis generates a safe set of atomic changes corresponding to the edit.

the test `Test.test1()` as affected.

In Jave, every field declaration in the body of an interface is implicitly `public`, `static`, and `final`, even if such fields are not explicitly declared that way. Our analysis will report a **CSFI**($I.START$) to represent this change. According to the Java language specification, references to compile-time constants must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field never cause initialization. In the example, the reference to `I.START` in method `A.foo()` is a reference to a field that is a compile-time constant; therefore, it does not cause interface `I` to be initialized. As a consequence, our analysis won't report change **CSI**($I.\langle clinit \rangle()$) in this case. Since change **CSFI**($I.START$) does not have a correlated method change, and our analysis needs to compare the call graph with method changes to locate the affected tests, the analysis fails to report this test as affected.

Even if we report the change **CSI**($I.\langle clinit \rangle()$) and correlate the **CSFI**($I.START$) change with it, our analysis would still fail to report the test as affected. The reason is that the real call graph of `Test.test1()` does not include the call to the method $I.\langle clinit \rangle()$.

The failure of the test in the edited program is caused by the value change to the variable `int i` in method `A.foo()`, which refers to a compile-time constant `I.START`.[12] But there is no syntactic dependence between this value change and the field initializer change in interface `I`. To capture this affected test, we need to extend our change impact analysis framework to model this kind of semantic dependence.

2.5.2 Changes to Import Statements

Suppose there are two library classes providing the same interfaces but different implementations, the programmer can change the import statements to use different libraries in the original and edited programs , but leave all the other part of the code untouched. In this case, a test's behavior may be affected because of the usage of different libraries. Whether we can detect the affected test or not depends on how the library class is used

[12] Whether the compile-time constant is defined in an interface or in a class does not affect the results.

in the program. If some user-defined type is a subtype of the imported library class, when we compare the type hierarchy of two versions of the program, the changes to import statements will be reflected as some dynamic dispatch changes. However, if the library class is not extended, but just referred as the type of some variables or return type of the methods, the type hierarchies in two different versions of the program will remain the same and our analysis won't report any atomic changes and thus fail to detect the affected test.

Chapter 3

Chianti– A Tool for Change Impact Analysis of Java Programs

To demonstrate the utility of the basic change impact analysis framework described in Chapter 2, we designed and implemented a proof-of-concept prototype–*Chianti*, a change impact analysis tool for Java program that has been integrated closely with Eclipse [18], a widely used open-source development environment.

Later in this chapter, we describe its validation against the 2002 revision history (taken from the developers' CVS repository) of *Daikon*, a realistic Java system developed by M. Ernst *et al.* [21, 22]. Essentially, in this initial study we substituted CVS updates obtained at intervals throughout the year for programmer edits, thus acquiring enough data to make some initial conclusions about our approach. We present both data measuring the overall effectiveness of the analysis and some case studies of individual CVS updates.

3.1 Prototype

Chianti has been implemented in the context of the Java editor of Eclipse, a widely used extensible open-source development environment for Java. Our tool is designed as a group of Eclipse plugins, *Chianti core*, a *launch configuration* and a *Chianti results view*. *Chianti core* is responsible for deriving a set of atomic changes from two versions of an Eclipse project (i.e., Java program), which is achieved via a pairwise comparison of the abstract syntax trees of the classes[1] in the two project versions; and obtaining affected

[1] While Eclipse provides functionality for comparing source files at a textual level, we found the amount of information provided inadequate for our purposes. In particular, the class hierarchy information provided by Eclipse 3.1 does not currently include anonymous and local classes.

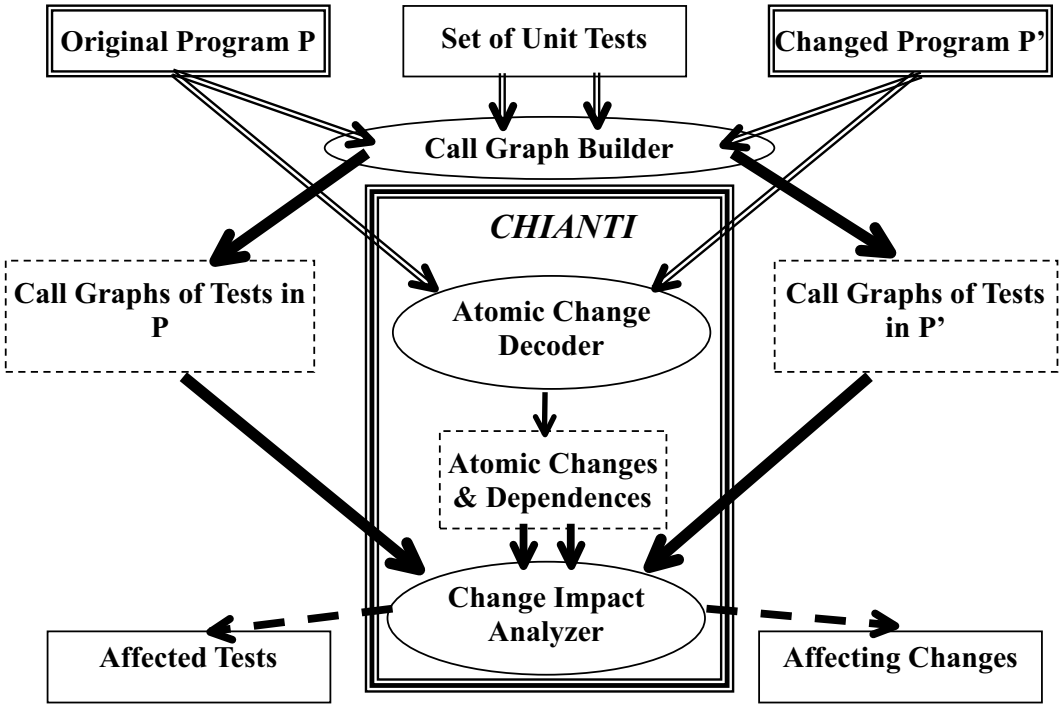


Figure 3.1: *Chianti core* architecture.

tests and their affecting changes by traversing the test's call graph in the original and edited programs. Figure 3.1 depicts *Chianti core* architecture. The *launch configuration* plugin allows users to select the project versions to be analyzed, the set of tests associated with the project, and the call graphs to be used. This provides programmers the flexibility of using call graphs generated by other tools as long as the call graph format matches the interface *Chianti* requires. The *Chianti results view* plugin manages the views that allow the user to visualize change impact information. The *Chianti launch configuration* provides an extension point allowing different programmers to implement their own preferred views for visualization.

Crisp is a tool for constructing intermediate versions of a Java program, which will be described in detail in Chapter 5. Like *Chianti*, *Crisp* is built as an Eclipse plug-in. *Crisp* shares the same *launch configuration* with *Chianti*, and calls the functionalities provided by *Chianti core* to generate the atomic changes for two versions of a Java program, as well as the affecting changes of an affected test. *Crisp* extends the *Chianti results view* to allow a programmer to select certain affecting changes for an affected

test, and build an intermediate program based on the programmer's selection.

Although *Chianti* is intended for interactive use, we have been testing the prototype using successive CVS versions of a program. Thus, a typical scenario of a *Chianti* session begins with the programmer extracting two versions of a project from a CVS version control repository into the workspace. The programmer then starts the change impact analysis launch configuration, and selects the two projects of interest as well as the test suite associated with these projects. Currently, we allow tests that have a separate `main()` routine and *JUnit* tests [32].

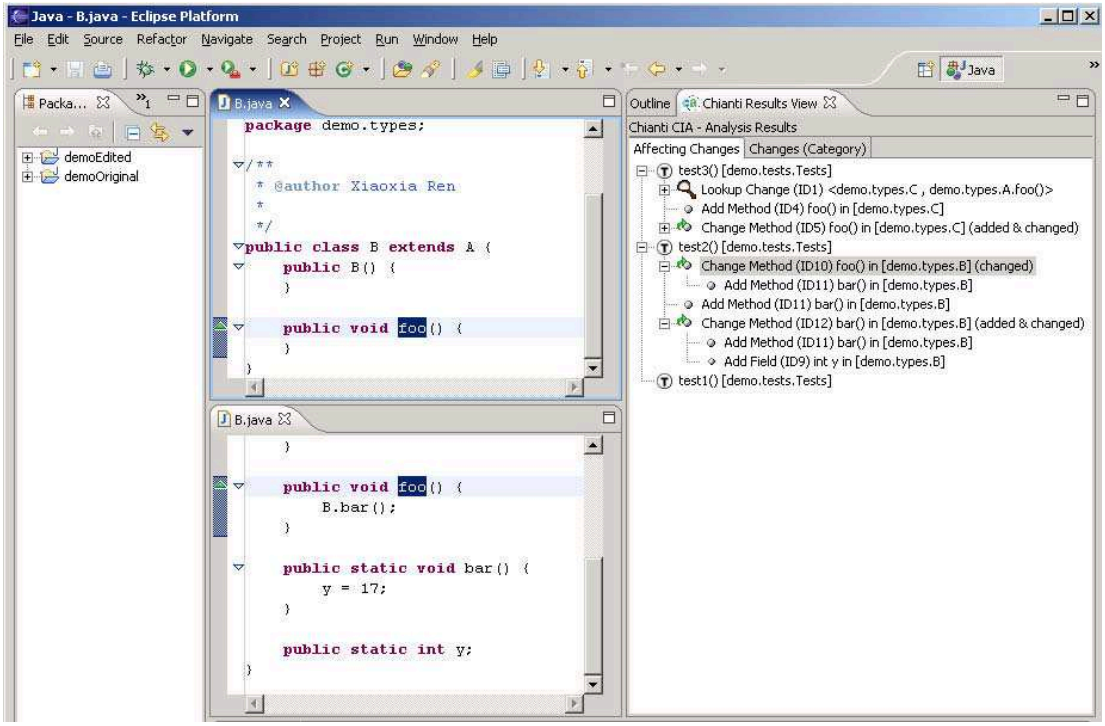


Figure 3.2: Snapshot of *Chianti*'s *affecting changes* view

In order to enable the reuse of analysis results, and to decouple the analysis from GUI-related tasks, both atomic change information and call graphs are stored as XML files. *Chianti* currently supports two mechanisms for obtaining the call graphs to be used in the analysis. Users can use the *Chianti* built-in call graph builder to generate the call graph automatically. In this case, *Chianti* will use *jikesBT*[2] to instrument

[2] jikes Bytecode Toolkit is a Java class library which enables Java programs to create, read, and write binary Java class files, and to query and update a single high-level representation of the collection of them, including relationships among them. This allows developing tools which report on what APIs the

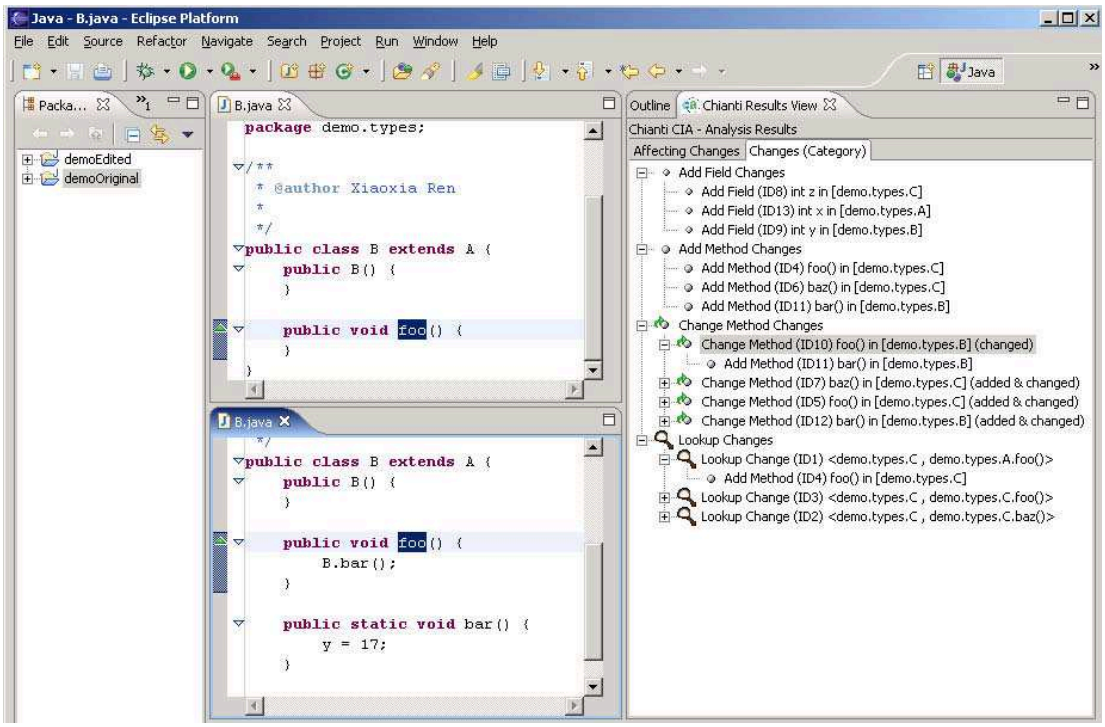


Figure 3.3: Snapshot of *Chianti*'s *changes by category* view

the Java binary class file and then execute the instrumented code to obtain the test's dynamic call graph.[3]

In its previous versions, *Chianti* built static call graphs by invoking the *Gnosis* analysis engine[4] to construct these [49]. Users needed to supply some additional information relevant to this analysis engine (e.g., the choice of call graph construction algorithm to be used and some policy settings for dealing with reflection). Now we have discontinued this option for users.

In addition, users can point *Chianti* directly at an XML file representation of the call graphs that are to be used, in order to enable the use of call graphs that have been constructed by external tools.

---

classes use, reorder and change instructions, merge or extend classes, add customized instrumentation (similar to profilers), analyze control and dataflow, etc.

[3] We did not optimize the gathering of the dynamic call information; presently, the instrumented tests run, on average, about 2 orders of magnitude more slowly than uninstrumented code, but we think we can reduce this overhead significantly with some effort.

[4] *Gnosis* is a static analysis framework that has been developed at IBM Research as a test-bed for research on demand-driven and context-sensitive static analysis.

When the analysis results are available, a new view *Chianti Results View* will stack on top of the outline view in the Java perspective. Since *Chianti* is expected to be used during programming and debugging, we integrate it into existing Java perspective rather than define a new perspective. The *Chianti Results View* provides users two ways of traversing the analysis results.

- The *affecting changes view* shows all tests in a tree view. Each affected test can be expanded to show its set of *affecting changes* and their prerequisites. Figure 3.2 shows a snapshot of this view; note how the prerequisite changes are shown. Each atomic change is the root of a tree that can be expanded on demand to show prerequisite changes. This quickly provides an idea of the different "threads'' of changes that have occurred.

- The *atomic-changes-by-category view* shows the different atomic changes of the edit grouped by category. The atomic changes and dependences shown in this view are not related to any specific tests, it just summarizes the results of comparison of two versions of a Java program. Figure 3.3 shows a snapshot of this view.

Each of these user interface components is seamlessly integrated with the standard Java editor in Eclipse (e.g., clicking on an atomic change in the *affecting changes view* opens an editor on the associated program fragment).

3.2 Evaluation

The experiments[5] with *Chianti* were performed on versions of the Daikon system by M. Ernst *et al.* [21, 22], extracted from the developers' CVS repository. The Daikon CVS repository does not use version tags, so we partitioned the year-long version history arbitrarily at week boundaries. All modifications checked in within a week were considered to be within one *edit* whose impact was to be determined. However, in cases where no editing activity took place in a given week, we extended the interval by one week

[5]All the data shown in this section were obtained in March 2004, and at that time, atomic change **CTD** was not defined yet in *Chianti*.

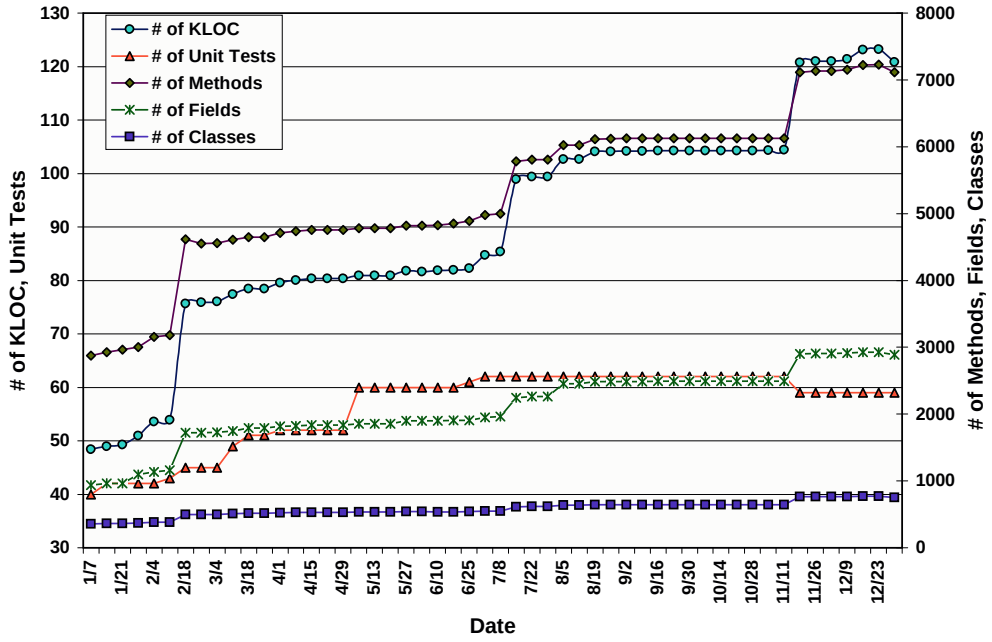


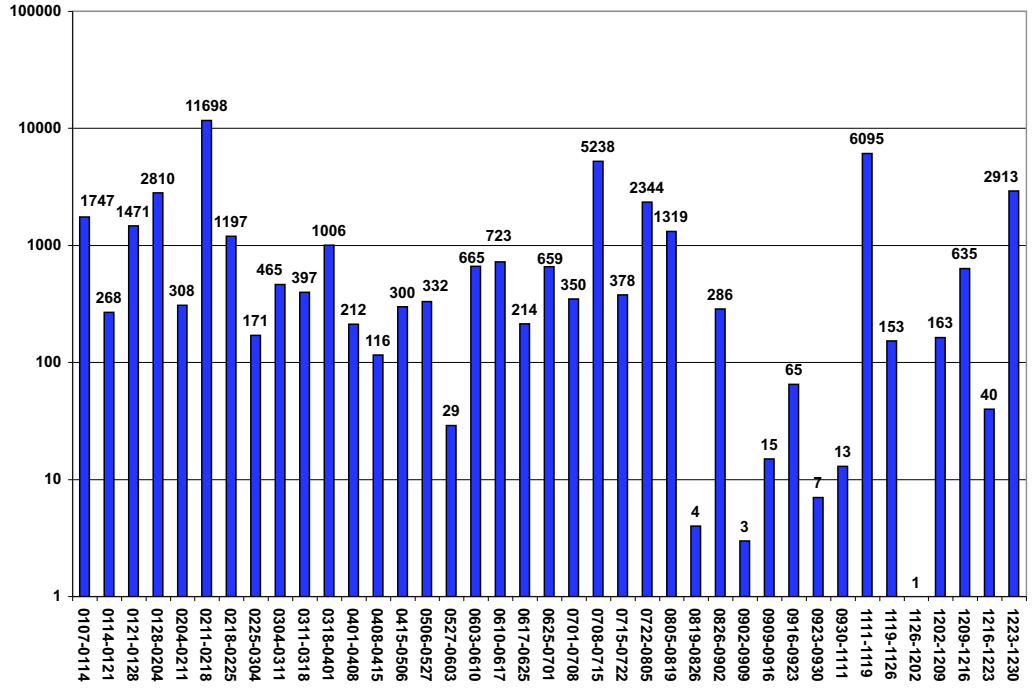
Figure 3.4: Daikon growth statistics for the year 2002

until it included changes. The data reported in this section covers the entire year 2002 (i.e., 52 weeks) of updates, during which there were 39 intervals with editing activity.

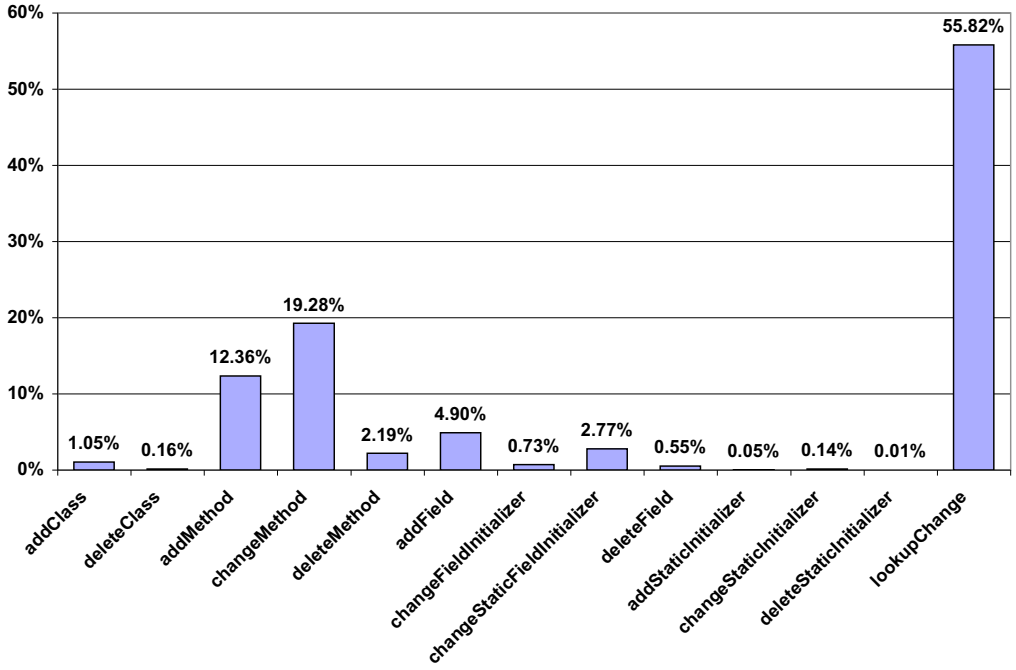
During the year under consideration, Daikon was actively being developed and increased in size from 48K to 123K lines of code. More significant are the program-based measures of growth, from 357 to 755 classes, 2878 to 7112 methods, and 937 to 2885 fields. The number of unit tests associated with Daikon grew from 40 to 62 during the time period under consideration. Figure 3.4 shows in detail the growth curves over this time period. Clearly, this is a moderate-sized application that experienced considerable growth in size (and complexity) over the year 2002.

3.2.1 Atomic Changes

Figure 3.5(a) shows the number of atomic changes between each pair of versions. The number of atomic changes per interval varies greatly between 1 and 11,698 during this period, although only 11 edits involved more than 1,000 atomic changes. Section 3.2.3 gives more details about two specific intervals in our study. Investigation of the largest edit revealed that during this week a parser was added to the system, which involved the addition of 100+ classes. The largest edit represents a redesign step that altered



(a)



(b)

Figure 3.5: **(a)** Number of atomic changes between each pair of Daikon versions in 2002 (note the log scale). **(b)** Categorization of the atomic changes, aggregated over all Daikon edits in 2002.

most of the system.

Figure 3.5(b) summarizes the relative percentages of kinds of atomic changes observed during 2002. The height of each bar indicates the frequency of the corresponding kind of atomic change; these values vary widely, by three orders of magnitude. Three of our atomic change categories were not seen in this data, namely *addInitializer, changeInitializer* and *deleteInitializer*. This is not surprising because, in Java, instance initializers are only needed in the rare event that an anonymous class needs to perform initialization actions that cannot be expressed using field initializers. In non-anonymous classes, it is generally preferable to incorporate initialization code in constructors or in field initializers. Note that the 0.01% value for *deleteStaticInitializer* in the figure represents the 5 atomic changes of that type out of a total of over 44,000 changes for the entire year!

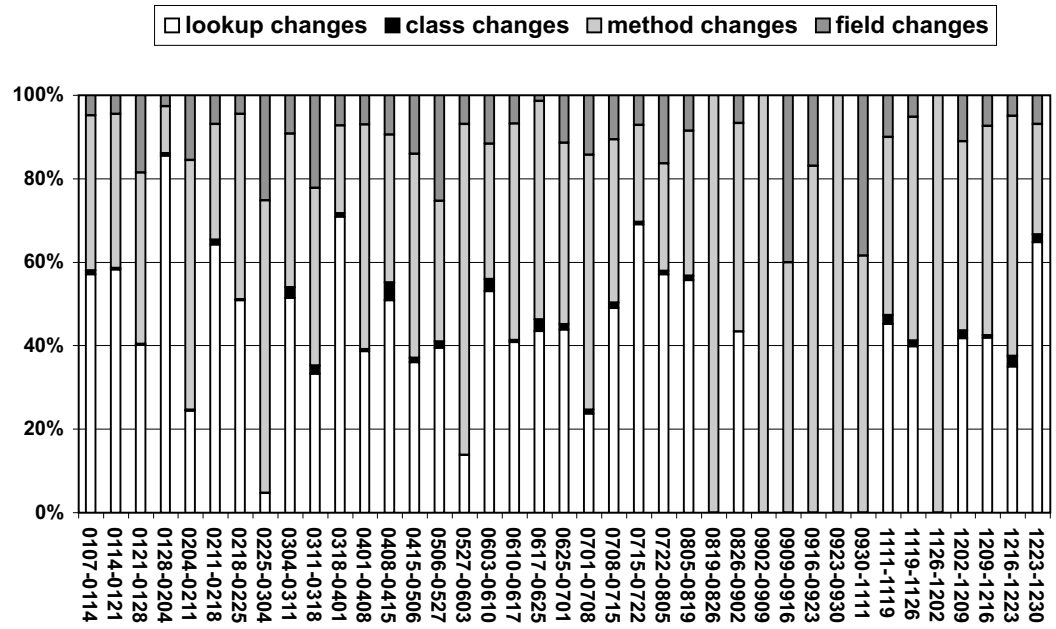


Figure 3.6: Classification of atomic changes for each pair of versions. Class changes include **AC** and **DC**. Method changes include **AM**, **CM**, **DM**, **ASI**, **DSI** and **CSI**. Field changes include **AF**, **DF**, **CSFI** and **CFI**.

Figure 3.6 shows the proportion of atomic changes per interval, grouped by the program construct they affect, namely, classes, fields, methods and dynamic dispatch.

Clearly, the two most frequent groups of atomic changes are changes to dynamic dispatch (i.e., **LC**) and changes to methods (i.e., **CM**); their relative amounts vary over the period.

3.2.2 Affected Tests and Affecting Changes

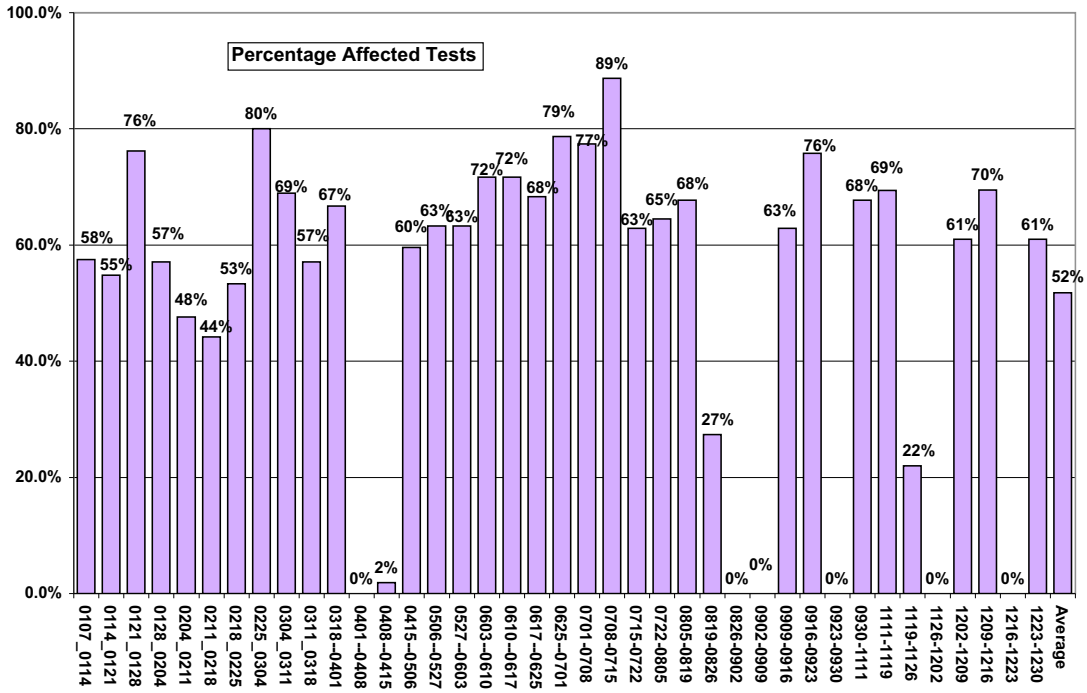


Figure 3.7: Percentage of affected tests for each of the Daikon versions.

Figure 3.7 shows the percentage of affected tests for each of the Daikon versions. On average, 52% of the tests are affected in each edit. Interestingly, there were several intervals over which no tests were affected, although atomic changes did occur. For example, there were no affected tests for the interval between 04/01/02 and 04/08/02, despite the fact that there were 212 atomic changes during this time. Similarly, for the interval between 8/26/02 and 9/02/02 there were 286 atomic changes, but no affected tests. This means that the changed code for these intervals was not covered by any of the tests! In principle, a change impact analysis tool could inform the user that additional unit tests should be written when an observation of this kind is made.

Figure 3.8 shows the average percentage of affecting changes per affected test, for

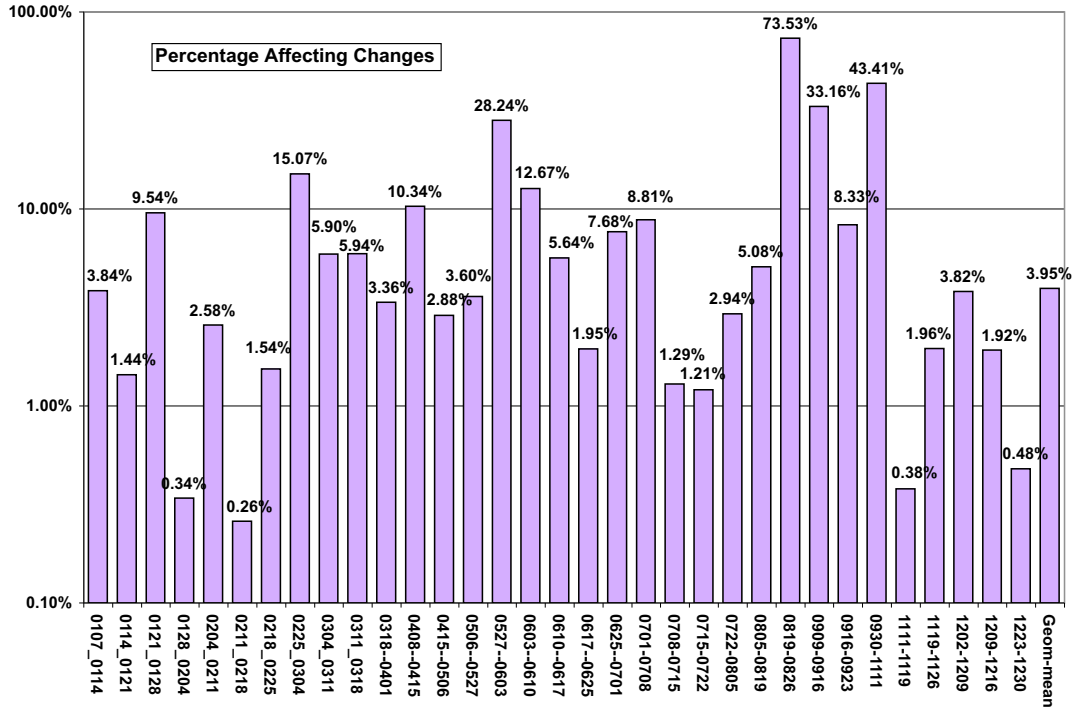


Figure 3.8: Average percentage of affecting changes, per affected test, for each of the Daikon versions. Note the logarithmic scale.

each of the Daikon versions. On average, only 3.95% of the atomic changes impact a given affected test. This means that our technique has the potential of dramatically reducing the amount of time required for debugging when a test produces an erroneous result after an editing session.

By contrast, the study performed with *Chianti* using static call graphs for the same Daikon data, yielded on average 56% affected tests and 3.7% affecting changes per affected test [49].[6] The closeness of these results to those reported in the present thesis suggests that we should investigate the trade-offs associated with using static or dynamic call graphs.

Our approach assumes that the test suite associated with a Java program offers good coverage of the entire program. To verify this assumption, we used the *JCoverage* tool (see `www.jcoverage.com`) to determine how many methods in Daikon were actually

[6] Imprecision in the static call graphs resulted in the detection of extra affected tests that had relatively small numbers of affecting changes. This skewed our averaging calculations to yield the counterintuitive result that the affecting changes percentage obtained using static call graphs was lower than the percentage obtained using the more precise dynamic call graphs.

exercised by its unit test suite. For each version of Daikon, we obtained the number of methods covered by the associated tests and the total number of (source code) methods in that version, yielding an average method coverage ratio. The overall average of these ratios on the entire Daikon system is quite low at 21%. However, this number is skewed by the fact that certain Daikon components have reasonable coverage (e.g., for the `utilMDE` component we find an average coverage ratio over the year of 47%), whereas other components (e.g., the `jtb` component) have virtually no coverage. Thus, while our change impact analysis findings are promising, they would be more compelling with a test suite offering better coverage of the system.

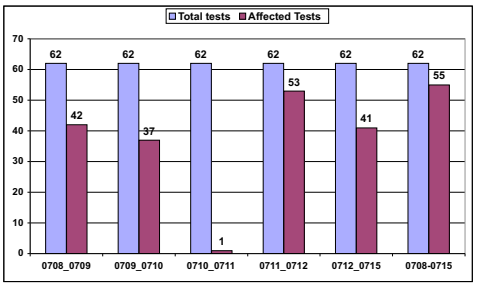
3.2.3 Case Studies

We conducted two detailed case studies to further investigate the possible applications of *Chianti* as it is intended to be used, namely in interactive environments with short time intervals between versions. To this end, we selected two one-week intervals from the whole year's data in which heavy editing activity occurred, and divided those intervals into subintervals of one day each.

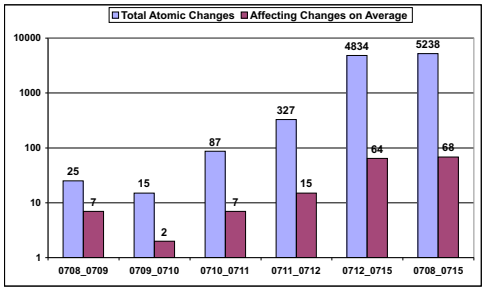
Case Study 1 The first interval we decided to explore further is the one for which we found the highest percentage of affected tests. This occurred between versions 07/08/02 and 07/15/02, when 88.7% (55 out of 62) of the tests were affected. We partitioned the version history of this interval into daily intervals so that we could obtain changes with finer granularity. In cases where no editing activity took place between two days, we extended the interval by one day, thus obtaining 5 intervals with editing activity.

Figure 3.9(a) shows the number of affected tests for each subinterval as well as the number of affected tests for the original week-long interval (shown as the rightmost pair of bars). Before partitioning, 55 of the 62 unit tests were affected tests, but smaller numbers of affected tests, ranging from 1 to 53, were reported for each of the subintervals (for example, in subinterval 07/10/02—07/11/02, there is only one affected test).

Figure 3.9(b) shows the total number of atomic changes and the average number of affecting changes per affected test in each subinterval compared with the original interval (again shown as the rightmost pair of bars). The use of smaller intervals



(a) Number of affected tests on large and smaller daily intervals.



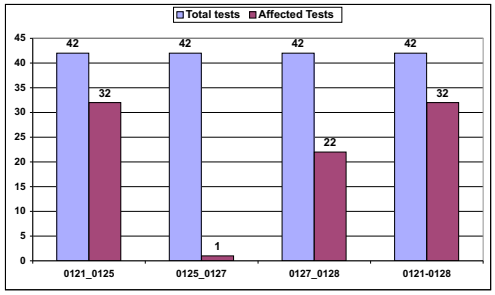
(b)Number of atomic changes and on average affecting changes on large interval and smaller daily intervals (note log scale)

Figure 3.9: Detailed analysis results for Daikon interval 7/08/02—7/15/02.

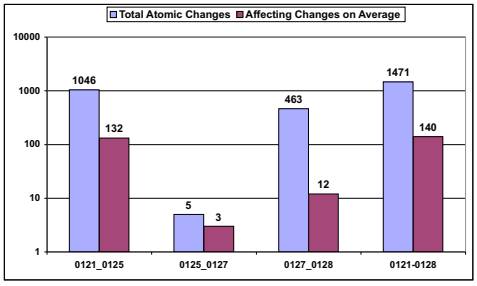
resulted in smaller numbers of atomic changes for each interval and also smaller numbers of affecting changes per affected test; this makes the tracing of the effects of affecting changes much easier. In addition, we found that 12 of the 55 affected tests for the original, week-long interval were only affected in one of the smaller intervals, which means that we can narrow down the range of affecting changes into a small set of atomic changes for these 12 tests.

Case Study 2 The second interval we selected is the one with the highest average number of affecting changes. This interval took place between versions 01/21/02 and 01/28/02, when 140 affecting changes occurred on average (ranging from 3 to 217) for 32 affected tests. Similar to case study 1, we partitioned the original week-long interval into several subintervals, obtaining 3 subintervals with editing activity. In Figure 3.10(a) and (b) we can see similar results to those of case study 1, that is, we obtained smaller numbers of atomic changes, affected tests and affecting changes compared to the original week interval.

In both case studies, we found that the use of subintervals with smaller numbers of affecting changes improved the ability of *Chianti* to help programmers with understanding the effects of an edit. Even in subintervals such as 01/21/02—01/25/02, where the number of atomic changes and the average number of affecting changes are large relative to the corresponding numbers for the original interval, *Chianti* can provide useful insights. For example, consider one test with a large number of affecting



(a) Number of affected tests on large and smaller daily intervals.



(b)Number of atomic changes and on average affecting changes on large interval and smaller daily intervals (note log scale)

Figure 3.10: Detailed analysis results for Daikon interval 1/21/02—1/28/02.

changes: `daikon.test.diff.DiffTester.testPpt4Ppt4` from subinterval 01/21/02—01/25/02. The affecting changes for this test are: 67 **CM** changes, 67 **AF** changes, and 69 **CSFI** changes. Among the 67 **CM** changes, 65 of them are associated with static initializers for some class. These, in turn, are dependent on 68 of the **CSFI**s, whose own prerequisites are 66 of the **AF**s. A closer look revealed that all the added fields have the same name, *serialVersionUID*, which is used to add serialization-related functionality to Daikon. It is interesting to observe that *Chianti* was able to determine that the changed behavior of this test was almost entirely due to this serialization-related change, and that the other 800+ atomic changes that occurred during this interval did not contribute to the test's changed behavior.

3.2.4 Chianti Performance

The performance of *Chianti* has thus far not been our primary focus, however, we have achieved acceptable performance for a prototype. Deriving atomic changes from two successive versions of Daikon takes, on average, approximately 87 seconds [48]. Computing the set of affected tests for each version pair takes approximately 5 seconds on average, and computing affecting changes takes on average approximately 1.2 seconds per affected test. All measurements were taken on a Pentium 4 PC at 2.8Ghz with 1Gb RAM.

Chapter 4

Dependences between Atomic Changes

Chapter 3 introduces our change impact analysis tool– *Chianti*, and how we obtain the affected tests and affecting changes for a specific test. The experimental results show that on average, the set of affecting changes of an affected test is small relative to the total number of atomic changes. However, examining each of these changes and pinpointing the few that induce the failure of a test is still a tedious task if performed manually. For large applications, the 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 specific test.

To further isolate relevant portions of an edit that directly cause the failure of a regression test, we should allow the user to select the suspected changes and apply them to the original program to create intermediate program versions. The resulting intermediate programs can then be tested using the tests that failed earlier. Programmers can ignore those 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 this process, in which the programmer does not need to be concerned with the syntactic inter-relationships of the changes, nor with manually editing any code.

Atomic changes have syntactic inter-dependences which induce a partial ordering $\prec$ on a set of them, with transitive closure $\preceq^*$. $C_1 \prec C_2$ denotes that C_1 is a prerequisite for C_2, as we described in Chapter 2. 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.

In our analysis, three kinds of dependences are defined between atomic changes, and they are all syntactic dependences that must be satisfied to ensure compilability. In this chapter, first we will show by example how to utilize the syntactic dependences to construct intermediate programs to help locate the failure-inducing changes, then we will discuss three kinds of dependences one by one in detail and the limitations of our current dependence graph.

4.1 Overview of Approach for Locating Failure-Inducing Changes

4.1.1 The Example Program

We will use the example program of Figure 4.1 to illustrate the dependences and how we locate the failure-inducing changes.[1] Figure 4.1(a) 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 [32], `Tests.test1`, `Tests.test2`, and `Tests.test3`, which are shown in Figure 4.1(b). Note that it is assumed that these tests will be used with both versions of the program.

Figure 4.1(c) shows the atomic changes corresponding to the two versions of the example program, numbered 1 through 12 for convenience. An arrow from an atomic change A_1 to an atomic change A_2 indicates that A_1 is dependent on A_2.

Figure 4.1 (d) shows the dynamic call graphs for the 3 tests `test1`, `test2`, and `test3`, before the changes have been applied. We can easily decide 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 4.1(e). Only call graphs for `test2` and `test3` are needed, since `test1` is not affected by any of the changes. We can compute the affecting changes for `test3`

[1] The example was first used in [45].

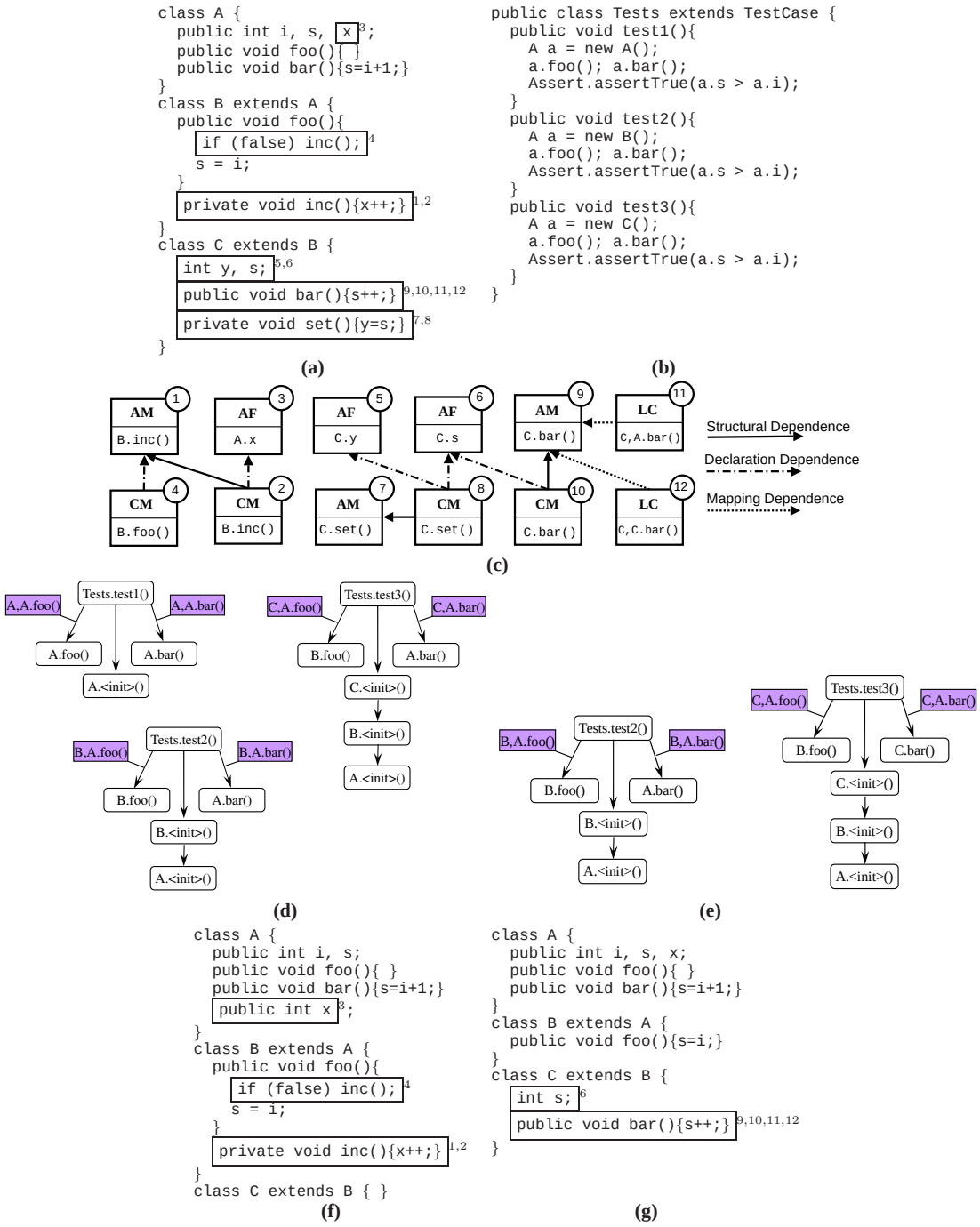


Figure 4.1: **(a)** Original and edited version of example program. **(b)** Tests associated with the example of (a). **(c)** Atomic changes for the example program, with their inter-dependences. **(d)** Call graphs for the tests in the original program version in (a). **(e)** Call graphs for the tests 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.

as follows. Observe, that the call graph for `test3` in Figure 4.1(e) contains methods `B.foo()`, 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. 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, 4, 6, 9, 10 and 11. Similarly, `test2` is affected by atomic changes 1 and 4.

4.1.2 Locating Failure-Inducing Changes by Constructing Intermediate Programs

The original program passed all the tests, but `test3` failed in the edited version. As Figure 4.1(c) 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 6 changes are the likely reason(s) for the test failure?* 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 following the dependence graph, we can generate an valid intermediate program which extends the original program with the selected atomic changes as well as all the other necessary prerequisite atomic changes.

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, following the dependence graph shown in Figure 4.1(c), atomic change 1 (i.e., *AM(B.inc())*) should also be included in the set to apply. In addition, to maintain the semantics of the program, we always extend 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 set to apply. Thus selecting atomic change 4 results in applying atomic changes 1, 2, 3 and 4 to create the intermediate program P_1 shown in Figure 4.1(f). Notice that the affecting changes set of `test3` does not include atomic change 2 and 3, but we apply these two changes to make it possible to generate valid (i.e., compilable) intermediate programs. Fortunately, since these augmented changes are not affecting

changes, they do not affect the test.

Note that programmers can select any affecting changes they want to inspect in any order. The dependences among atomic changes guarantee the validity of 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. She may then suspect that the newly added method `C.bar()` is the potential culprit in the edit. She can rollback and restart from the beginning and obtain another intermediate version P_2 shown in Figure 4.1(g) by applying atomic changes 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 on 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 a failure-inducing change.

With the help of intermediate programs, 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.

4.2 Structural Dependence

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

4.2.1 Addition and deletion of 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. 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 in Figure 4.1(c) 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 Figure 4.1 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, and therefore could not have affected its behavior.

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(C1LocalC)*.

4.2.2 Changing a field type or method return type.

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 with the same name but different field type. The dependence also defines the ordering of operations in implementation. If these two changes are applied in the reverse order, after the **AF** change is applied, the intermediate program includes two fields with the same name,

then it is not an easy job to apply the **DF** change.[2] Similar dependences are reported when the return type of a method is changed.

4.3 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 Figure 4.1(c), 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.

4.3.1 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 Figure 4.1(c), the dependences between pairs of atomic changes (1, 4), (3, 2), (5, 8), (6, 8), (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()`, or any `Exception` subtypes thrown by method `X.foo()`; *AC(A)* $\prec_{declaration}$ *CM(X.foo())* if type `A` is used in the changed method `X.foo()` in the edited version; *AC(A)* $\prec_{declaration}$ *CTD(C)* if in the edited program, type `C` is a subtype of type `A`. All the declaration dependences shown in these examples are related to the addition of a type declaration, And we can easily generalize these dependences to the additions of field and method declarations.

4.3.2 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.

[2]Ideally, these two fields can be distinguished by field types, but *Crisp* uses the standard function provided by Eclipse and it can only locate the fields by their names.

Consider the example shown in Figure 4.2, 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()`. Our analysis 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 (i.e., all the classes that implement `I`). Otherwise, adding only method `I.foo()` to the original program P will result in an intermediate program P'' which cannot compile.

```
interface I {

}

class A implements I {

}
```

(a) P

```
interface I {
   public void foo();
}
class A implements I {
   public void foo(){...}
}
```

(b) P'

Figure 4.2: Add method declaration to an interface **(a)** Original program P. **(b)** Edited program P', the new added code is shown in boxes.

A similar dependence occurs when the programmer changes the modifier of a method from *abstract* to non-abstract or *vice versa*. Consider the example program in Figure 4.3. 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`.

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

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

(a) P

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

(b) P'

Figure 4.3: 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.

4.3.3 Necessary method declarations for a class

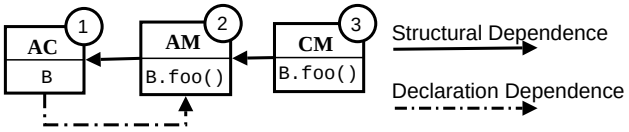
Usually we can declare an empty class without any members, but not always. In some cases, we must add some necessary methods to make the class compile.

Overriding methods

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

1

2,3



(a) Example program **(b) Dependence graph**

Figure 4.4: Necessary method declarations for a new class. **(a)** Original and edited version of example program. The added class and method are shown in a box. Each box is labeled with the numbers of the corresponding atomic changes. **(b)** Atomic changes and their dependences

One case is when programmers declare a class which extends an abstract class or implements an interface, then this new class cannot be empty, and must have some overriding methods implemented. Consider the example program in Figure 4.4(a) (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 4.2, two structural dependences

are reported: *AC(B)* $\prec_{structural}$ *AM(B.foo())* $\prec_{structural}$ *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)*. Figure 4.4(b) shows the dependence graph between the atomic changes. We observe that the structural dependences and declaration dependences between atomic change 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. If our analysis generates dependence cycles among several atomic changes, no matter which atomic change in the cycle is selected by the programmer, all the atomic changes in the cycle should be automatically collected and applied to the original program to construct the valid intermediate program.

Necessary constructors

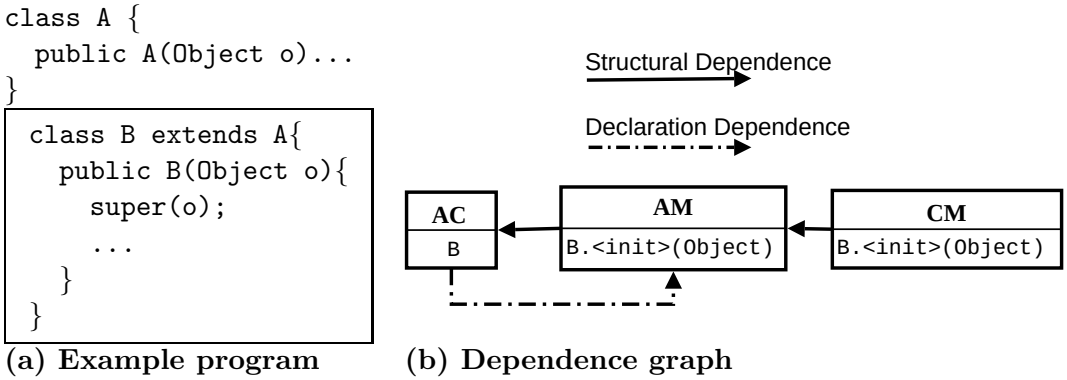


Figure 4.5: Necessary constructor declarations for a new class. **(a)** Original and edited version of example program. The added method is shown in a box. **(b)** Atomic changes and their dependences

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. Consider the example program shown in Figure 4.5. The original program consists of a class `A` with constructor `A(Object)` defined and the edited program declares a new class `B` which extends class `A` and class `B` defines a

constructor which calls method `super(..)` explicitly. In such cases, *Chianti* reports a declaration dependence *AM(B.B(Object))* $\prec_{declaration}$ *AC(B)*, which means that to declare the new class `B`, its constructor `B(Object)` must be added in the intermediate program. This declaration dependence forms a cycle with the structural dependence *AC(B)* $\prec_{structural}$ *AM(B.B(Object))*; this forces the intermediate program to always include the corresponding constructors when the class is added and guarantees the compilability of the intermediate program.

If there is more than one constructor with arguments defined in class `B`, our analysis will report declaration dependences between adding each constructor and the addition of the new class `B`. Although only one of the dependences is necessary to guarantee the validity of the intermediate program, our analysis reports the dependences conservatively because we did not find a good way of representing the relation that either *AM(B.m())* or *AM(B.f())* is the prerequisite of *AC(B)*, but not necessarily both.

4.4 Mapping Dependence

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.

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 4.1, 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 a node in the call graph and lookup changes (**LC**) that correspond 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*.

4.4.1 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 `C.⟨init⟩()` 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.⟨clinit⟩())*.

4.4.2 Field type or method return type changes

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

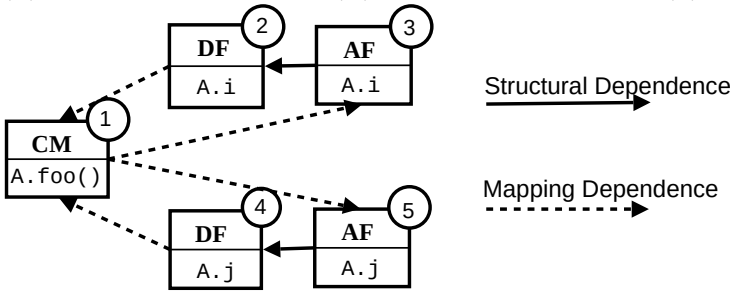
(a) P

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

(b) P'

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

(c) Tests



(d) Dependence graph

Figure 4.6: 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.

As we discussed in Section 4.2, 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 and reports those as changed elements, and creates the corresponding mapping dependences.

Considering the example program in Figure 4.6, 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 the field `i` and `j` are both changed to `String`. The output of `testA()` changes to `null` instead of `0` in the edited program, since the default value of a string is `null`. *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 although there aren't any code changes to this method, and *Chianti* also creates mapping dependences: $\{AF(A.i), AF(A.j)\} \prec_{mapping} CM(A.foo())$ and $CM(A.foo()) \prec_{mapping} \{DF(A.i), DF(A.j)\}$ shown in Figure 4.6.

Notice that these dependences form two dependence cycles, thus forcing all the five changes to be applied together to obtain the intermediate version, no matter which of the changes is selected by 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` can not be assigned to an `int`. This implies that all the changed fields are forced to be changed at the same time if they are all used in the same method or the same initializer block.

4.4.3 LC changes.

```
class A {
  public void foo()...
}
```

(a) P

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

(b) P'

Figure 4.7: Addition of a new class results in **LC** changes. **(a)** Original program P. **(b)** Edited program P', the new added code is shown in boxes.

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 run-time type `Y` results in the selection of a different method in the edited program. Many source code changes can be mapped to **LC** changes.

Addition or deletion of an overriding method may result in **LC** changes. Consider the example in Figure 4.1(a), a new overriding method `bar()` is added to class `C` in the edited program P'. Two mapping dependences are shown in Figure 4.1(c): *AM(C.bar())* $\prec_{mapping}$ *LC(C, A.bar())* and *AM(C.bar())* $\prec_{mapping}$ *LC(C, C.bar())*.

Addition or deletion of a class may also result in **LC**. Considering the example in Figure 4.7, the original program P includes a class `A` which defines method `A.foo()`. The programmer adds a new `"class B extends A"` with empty members in the edited program P'. This edit results in an **LC** change *LC(B, A.foo())*, since in the original program P, there is no dynamic lookup for run-time type `B`, while in the edited program P', programmers can invoke a call to method `A.foo()` on an object of type `B`. *Chianti* reports a mapping dependence: *AC(B)* $\prec_{mapping}$ *LC(B,A.foo())*[3] to indicate that this **LC** change is caused by the addition of class `B`.

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` non-abstract, also results 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.

4.5 CTD Related Dependences

A type hierarchy change involves more complicated dependences with other atomic changes and we will discuss these dependences in detail in this section.

---

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

4.5.1 Declaration Dependences

Overriding Methods

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

(a) P

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

(b) P'

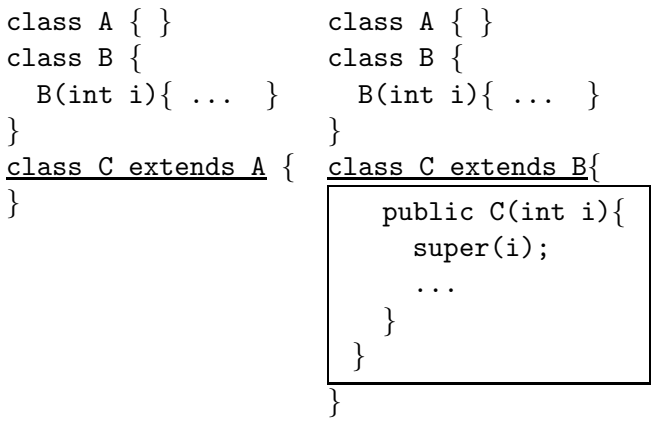
Figure 4.8: Type hierarchy changes requires addition of overriding methods. **(a)** Original program P. **(b)** Edited program P'.

When a programmer changes a class declaration to extend an abstract class or implement an interface, she has to implement all the abstract methods declared in the abstract super class or interface to make the edited class compile. Consider the example program in Figure 4.8. The original program consists of an abstract class `A` and the edited program changes the declaration of class `B` which extends class `A` and class `B` overrides method `A.foo()`. *Chianti* reports a declaration dependence: *AM(B.foo())* $\prec_{declaration}$ *CTD(B)*. If the programmer wants to apply the type declaration change, the overriding method must be added at the same time to guarantee the validity of the intermediate program.

Similarly, if the original superclass of a **CTD** change is an abstract class or interface, and if the programmer deletes the overriding methods in the edited program, we report the **CTD** change as the declaration prerequisite of the **DM** changes. If the programmer selects those **DM**s to apply to build the intermediate program, applying the type declaration change at the same time guarantees the compilability of the intermediate program.

Constructors

If the programmer changes a class declaration, the dependences between changes to constructors and a **CTD** are more complicated. Considering the example shown in



(a) P **(b)** P'

Figure 4.9: Type hierarchy changes requires addition of constructors. **(a)** Original program P. **(b)** Edited program P'.

Figure 4.9, in which the declaration of class `C` changes from `class C extends A` to `class C extends B`. Class `B` defines a constructor `B(int)` but no default constructors without parameters, and in the edited version, class `C` needs to explicitly define constructors (that have any kind of parameters or without parameters) which explicitly call `super(int)`. In this example, class `C` must define a constructor `C(int)` to make the edited program compile. Our analysis reports $\mathbf{AM}(C.\langle init\rangle(int)) \prec_{declaration} \mathbf{CTD}(C)$, meaning that to change the class `C`'s type declaration, we have to define an explicit constructor to make the resulting intermediate program compile.

Similarly, if we wanted to restore the edited program back to the original in Figure 4.9, that is change the declaration of class `C` from `class C extends B` to `class C extends A`, the constructor originally defined in class `C` needs to be deleted. Our analysis will report declaration dependence $\mathbf{CTD}(C) \prec_{declaration} \mathbf{DM}(C.\langle init\rangle(int))$, meaning that if the programmer wants to delete the definition of the constructor, she has to change the declaration of class `C` as well. We also report declaration dependence $\mathbf{CM}(C.\langle init\rangle(int) \prec_{declaration} \mathbf{CTD}(C))$, meaning that if the programmer wants to change the declaration of class `C`, she has to change the definition of its constructor. Otherwise, the constructor calls `super(int)`, but in the edited program, there isn't `A(int)` defined in class `A`.

Necessary method changes

```
class A{
  public String toString(){
    return "A";
  }
}
class B{
  public String toString(){
    return "B";
  }
}
class C extends B{ }
class D extends C{ }
class R{
  String foo(){
    B o = new D();
    return o.toString();
  }
}
```

(a)

```
class A{
  public String toString(){
    return "A";
  }
}
class B{
  public String toString(){
    return "B";
  }
}
class C extends A{ }
class D extends C{ }
class R{
  String foo(){
    A o = new D();
    return o.toString();
  }
}
```

(b)

```
class Test extend TestCase{
  public void testfoo(){
    Assert.assertTrue(new R().foo().equals("B"));
  }
}
```

(c)

Figure 4.10: Type declaration change results in other method changes **(a)** Original program. **(b)** Edited program. The edited code is shown in a box. **(c)** Tests correlated to the example program

When a programmer makes changes to a type declaration that results in a type hierarchy change, she may need to change the other part of the code to make the edited program compile. Considering the example shown in Figure 4.10, in which the declaration of class `C` changes from `class C extends B` to `class C extends A`. To make the edited program compile, the programmer needs to make the change to method `R.foo()` as shown in the box in Figure 4.10 (b). *Chianti* reports the dependence $CM(R.foo()) \prec_{declaration} CTD(C)$.

On the other hand, if for some reason, the programmer wanted to apply atomic

change *CM(R.foo())* to the original program to generate an intermediate program, then atomic change *CTD(C)* must also be applied, otherwise, the resulting intermediate program is invalid since the edited method `R.foo()` assumes that class `D` is a subtype of class `A`, which is not true if we don't apply atomic change *CTD(C)*. For this reason, we report dependences *CTD(C)* $\prec_{declaration}$ *CM(R.foo())*.

To generalize, we report *CTD(C)* $\prec_{declaration}$ *CM(R.m())* and *CM(R.m())* $\prec_{declaration}$ *CTD(C)* if the edited method `R.m()` calls the constructor of class `D` or refers to class `D` and `D` is a subtype of class `C`, or `D = C`. All the declaration dependences form a loop, which means that all the changes in the loop have to be applied at the same time.

4.5.2 Mapping Dependences

Virtual Method Changes

It is possible for changes to the hierarchy to affect the behavior of a method, although the code in the method is not changed. Considering the example program shown in Figure 4.11, In the original program, type C is declared as `class C extends B`, but in the edited program, the declaration changes to `class C extends A` and all the other type declarations remain the same, that is, the whole subtree rooted at class `C` is moved from subtree of class `B` to subtree of class `A`. Our analysis reports *CTD(C)* to represent this type declaration change of class C.

In the original program, both `testFoo()` and `testBar()` pass. However, in the edited program, `testFoo()` fails because `R.foo()` returns string "A", instead of string "B". Various constructs in Java such as `instanceof`, casts and exception `catch` blocks test the run-time type of an object. If such a construct is used within a method and the type lies in a different position in the hierarchy of the program before the edit and after the edit, then the behavior of that method may be affected by this hierarchy change (or restructuring). For this reason, `testBar()` throws a cast exception when object `o` is casted to type `B` in method `R.bar()`. Both tests are affected by the type declaration change to class `C`. To capture such affected tests correctly, we should find a way to map **CTD** changes to some method level changes.

```
class A{
  public String toString(){
    return "A";
  }
}
class B{
  public String toString(){
    return "B";
  }
}
class C extends B{ }
class D extends C{ }
class R{
  String foo(){
    Object o = new D();
    return o.toString();
  }
  String bar(){
    Object o = new D();
    return ((B)o).toString();
  }
}
```

(a)

```
class A{
  public String toString(){
    return "A";
  }
}
class B{
  public String toString(){
    return "B";
  }
}
class C extends A{ }
class D extends C{ }
class R{
  String foo(){
    Object o = new D();
    return o.toString();
  }
  String bar(){
    Object o = new D();
    return ((B)o).toString();
  }
}
```

(b)

```
class Test extend TestCase{
  public void testFoo(){
    Assert.assertTrue(new R().foo().equals("B"));
  }
  public void testBar(){
    Assert.assertTrue(new R().bar().equals("B"));
  }
```

(c)

Figure 4.11: Type declaration change results in mapping dependences **(a)** Original program. **(b)** Edited program. The edited code is shown in a box. **(c)** Tests correlated to the example program

If the programmer changes the type declaration of class `C` and results in a type hierarchy change, the initialization task of the class is changed, since its superclass or superinterface is changed. So our analysis reports **CM** changes to all constructors defined in class `C` (or to the default constructor created by the compiler), and also reports *CTD(C)* $\prec_{mapping}$ *CM(C.⟨init⟩(...))* for all these constructors to represent the dependences between **CTD** change and the virtual method changes to constructors. In this example, *CM(C.⟨init⟩())* is the change that causes both `testFoo()` and `testBar()` fail.

LC changes

CTD changes may also result in **LC** changes. Considering the example program shown in Figure 4.11, in the original program, we notice that for the callsite `o.toString()` in method `R.foo()`, the static method signature at this callsite is `java.lang.Object.toString()`, and the runtime type of object `o` reaching this statement is type `D`, the target callee method is `B.toString()`. But in the edited program, the target callee method changes to `A.toString()`, while the static method signature and the runtime receiver type at the callsite remain the same as in the original program. This target change is due to a dynamic method dispatch change *LC(D, java.lang.Object.toString())*, and the change to the type declaration of class `C` causes the **LC** change, so we report a mapping dependence *CTD(C)* $\prec_{mapping}$ *LC(D, java.lang.Object.toString())*. The other similar mapping dependences to **LC** changes in this example program include: *CTD(C)* $\prec_{mapping}$ *LC(C, java.lang.Object.toString())*; *CTD(C)* $\prec_{mapping}$ *LC(C, B.toString())*; *CTD(C)* $\prec_{mapping}$ *LC(D, B.toString())*; *CTD(C)* $\prec_{mapping}$ *LC(C, A.toString())*; and *CTD(C)* $\prec_{mapping}$ *LC(D, A.toString())*.

4.6 Limitations of Dependences

All the syntactic dependences discussed above are used for automatic construction of intermediate program versions; however, we found that the dependences are not

complete, that is, we are missing some dependences that are necessary to guarantee the validity of the intermediate program. This section discusses the details of the limitations of the dependences in our framework.

4.6.1 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, we have 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.

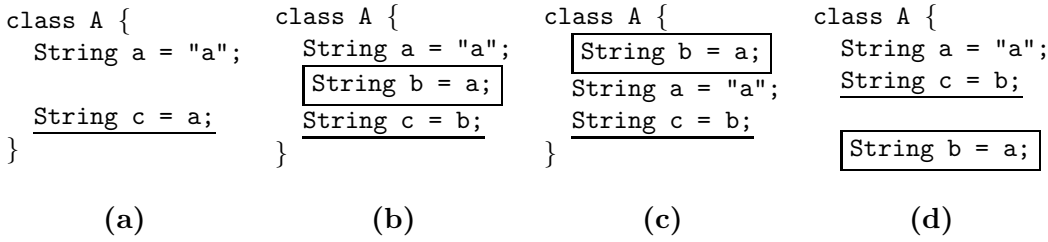


Figure 4.12: The positions of an added field affects the compilability of a program. (a) the original program P. (b) The edited program P'. (c) the first intermediate version P_1 by putting the new added field at the beginning of a class. (d) the second intermediate version P_2 by putting the new added field at the end of a class

Consider the example program in Figure 4.12, (a) and (b) represent the original and edited program respectively. The added field is shown in a box, and the changed field is underlined. There are three field changes: 1). *AF(A.b)*, 2). *CFI(A.b)* and 3). *CFI(A.c)*. Suppose the programmer wants to apply all three changes to new field `b` and existing field `c` to the original program. If we add field `b` at the beginning of the class, we obtain the intermediate program shown in Figure 4.12(c); it has a compilation error for field `b` because it refers to field `a` that is not yet declared. If we add field `b` at the end of the class, we get the intermediate program shown in Figure 4.12(d); this also has compilation error for field `c` because it refers to field `b` which is not yet declared.

In implementation, *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 1 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 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.

4.6.2 Value Changes.

The program behavior may depend on the values of some specific variables. In the example shown in Figure 4.13, a programmer defines a set of options in interface `Option`, and method A.foo() performs a different task based on the value of the obtained option.

The test in the original program passes and prints the selected option value. In the edited program, a new field `OPT2 = 1` is added to the interface and the programmer change the value of `OPT1` from `1` to `2`. The test in the edited program fails and throws a runtime exception. *Chianti* reports the following atomic changes and dependences: *CFI(Option.OPT1)* represents the change to the value of field `OPT1`; *AF(Option.OPT2)* $\prec_{structural}$ *CFI(Option.OPT2)* represents the addition and initialization of field `OPT2`; and declaration dependence *AF(Option.OPT2)* $\prec_{declaration}$ *CM(A.foo())* meaning that to make changes to method `A.foo()`, field `Option.OPT2` must be declared.

If the programmer selects *CM(A.foo())* to apply to the original program, its prerequisite change *AF(Option.OPT2)* is also applied, and *Crisp* automatically applies the field initializer change for this field *CFI(Option.OPT2)*. This results in the intermediate program shown in figure 4.13 (d). Because atomic change *CFI(Option.OPT1)* is not applied, the intermediate program has a compilation error, since two `case` statements share the same value (i.e., `OPT1` and `OPT2` are assigned the same value).

To solve this problem, we need to investigate more sophisticated dependences to represent the possible semantic dependences between atomic changes.

```
public interface Option {
  int OPT1 = 1;
}
public class A {
  public void foo() {
    int i = 1;
    switch (i) {
    case Option.OPT1:
      System.out.println(i);
      break;
    }
  }
}
```

(a)

```
public interface Option {
  int OPT1 = 2;
  int OPT2 = 1;
}
public class A {
  public void foo() {
    int i = 1;
    switch (i) {
    case Option.OPT1:
      System.out.println(i);
      break;
    case Option.OPT2:
      throw new RuntimeException("Error");
    }
  }
}
```

(b)

```
public class Test extends TestCase {
  public void test1(){
    new A().foo();
  }
}
```

(c)

```
public interface Option {
  int OPT1 = 1;
  int OPT2 = 1;
}
public class A {
  public void foo() {
    int i = 1;
    switch (i) {
    case Option.OPT1:
      System.out.println(i);
      break;
    case Option.OPT2:
      throw new RuntimeException("Error");
    }
  }
}
```

(d)

Figure 4.13: The value of an added field affects the compilability of a program. (a) the original program P. (b) The edited program P'. (c) The test case. (d) the intermediate program after applying change $\mathbf{CM}(A.foo())$

Chapter 5

An Application of the Change Dependence Graph

The goal of the syntactic dependences discussed in Chapter 4 is to guarantee the validity of the intermediate program versions. In this chapter, we first introduce *Crisp*, a tool built by Ophelia Chesley to semi-automatically construct the intermediate programs using the dependences graph. Then we present two case studies[1] to show that in most cases, our dependences graph help constructing valid intermediate programs and locating the failure-inducing changes for the real world Java programs.

5.1 Constructing Intermediate Program Versions

As a companion tool to *Chianti*, Ophelia Chesley developed *Crisp* [12, 45, 13] 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 core* for two versions of a Java program, as well as the affecting changes of an affected test. The major tasks that *Crisp* performs are (i) to gather and order all the prerequisites of the affecting changes and to create their *to-be-applied* set and present it, and (ii) 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 [18] environment provides APIs for accessing the abstract syntax trees of the original and the edited 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

[1]The case studies were published in [45]

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* applies atomic changes.

The ultimate goal in *Crisp* is to create a compilable intermediate program for each affecting change that the programmer selects. The three categories of dependences defined in Chapter 4 are used by *Crisp* to collect all the necessary prerequisites to be applied to the original program, and the *to-be-applied* set of a selected change. For declaration and mapping dependences, the ordering of applying these changes is arbitrary as long as all the changes in the set are present in the final intermediate program. For example, in Figure 4.1, $AM(B.inc()) \prec_{declaration} CM(B.foo())$, the choice of which of these two methods to apply first won't affect the compilability of the intermediate program.

However, certain orderings of structural dependences are critical to the process of *Crisp* creating a valid intermediate version. The *to-be-applied* set is therefore partially ordered. Within *Crisp*, structural dependences can be divided into two categories: (i) *pure structural* dependences and (ii) *buddy* dependences.

Pure Structural Dependences. Most of the structural dependences belong to this category and *Crisp* must handle 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. As mentioned before, *Chianti* decomposes some edits into several atomic changes; for example, it 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** as separate may be confusing to programmers, since they have to be applied together. 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 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: (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.

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** and 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 4.1, Figure 4.1(f) 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`. Fortunately, those newly added atomic changes won't affect the result of the test, which means that those changes are only added for the purpose of compilation. For example, *CM(B.inc())* is not in the affecting changes set of `test3`, and this method is not executed in the intermediate program.

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

Table 5.1: The sizes of case study data

5.2 Case Studies

The goal of the syntactic dependences is to guarantee the validity of the intermediate program versions given any user-selected atomic changes to apply to the original program. To evaluate the practicability, we conduct two case studies to check whether our dependences graph help constructing valid intermediate programs and locating the failure-inducing changes for the real world Java programs.

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 a Java project which 1) is at least moderate size, since bugs in small programs can be found easily using traditional debugging tools; 2) provides access to all the source code and development history; 3) has comprehensive unit tests and/or regression tests associated with the program including some that are failing tests.

In this section we present two case studies using CVS repository data from *Daikon* [21, 22] and *Eclipse jdt compiler* [18]. Table 5.1 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.

5.2.1 Daikon unit tests

In Chapter 3, we had extracted 52 weeks of Daikon [21, 22] source code from a CVS repository for testing the effectiveness of *Chianti* (i.e., all year 2002 check-ins). We did find some failures while executing these tests, however, those were always associated with the initial introduction of the test or with existing tests that always fail. But in

our experiment, we want to locate the failure-inducing changes for some test; that is, we need to find some test which passes in the previous version, but fails in the edited version. So 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 November 11^{th} version, but failed in the edited version dated November 19^{th}. *Crisp* called the functionality provided in *Chianti core* 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. Then *Crisp* created the to-be-applied sets for the affecting changes and presented them to the programmer. Our goal was to use *Crisp* to locate the changes that had caused the failure of these tests.

The results are summarized in Table 5.2. *Chianti* calculates 6093 atomic changes between these two versions of Daikon. For `testXor`, there are 35 affecting changes and using *Crisp* we found 2 failure-inducing changes. Similarly, there are 34 affecting changes for `testMinus` and with *Crisp*, we located 1 failure-inducing change. For both tests, the to-be-applied sets are exactly the same as the affecting changes set.

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

Table 5.2: Using *Crisp* for the Daikon versions Nov 11^{th} and Nov 19^{th}(2002).

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. For test `testMinus`, we were able to locate an atomic change *CM(daikon.diff.-MinusVisitor.shouldAdd(..))* whose application to the original program caused failure.

Index	Version pair	Atomic changes	Failing tests	Avg. affecting changes	Avg. failure-inducing changes
1	31Mar2003–01Apr2003	370	1	42	1
2	13Aug2003–14Aug2003	101	3	1	1
3	10Apr2004–11Apr2004	146	46	37	1
4	16Nov2004–17Nov2004	465	4	15	1
5	21Jan2003–22Jan2003	724	7	151	3
6	22Jan2003–23Jan2003	723	2	6	1
7	14Feb2003–15Feb2003	762	2	11	2
8	24Jun2003–25Jun2003	156	1	3	1

Table 5.3: Applying *Crisp* on Eclipse jdt core versions with failing tests.

For test `testXor`, we located 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 re-executed 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, 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.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`.

5.2.2 Eclipse jdt compiler unit tests

We performed our case study on an *Eclipse* plug-in project, `org.eclipse.jdt.core`. `jdt core` is the plug-in that defines the core Java elements and API. This plug-in includes an incremental Java compiler, a Java model that provides API for navigating the Java element tree, and other packages. Several test plug-ins are associated with `jdt core` and we chose the `org.eclipse.jdt.core.tests.compiler` plug-in to do the case study because of its availability with the development history of `jdt.core`. In addition, the tests in this plug-in are all *JUnit* tests.

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 `jdt.core.tests.compiler` plug-in, attempting to find failing tests. We succeeded in finding 4 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 5.3. 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 4 more version pairs in 2003 which are indexed 5 to 8 in Table 5.3.

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 5.3. In addition, we selected one version pair with the highest number of affecting changes for which we present further details of our findings in Table 5.4.

In the case study, we noticed that many of these failure-inducing changes cause failures in multiple tests which have similar affecting changes sets. In Table 5.4, *AM(CompletionParser.consumeClassHeaderName())* causes four tests to fail. 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.

Index	Failed Tests	# of Affecting Changes	Failure-inducing Changes
1	CompletionParserTest.testZA_1	142	CM CompletionParser.consumeEnterVariable()
2	CompletionParserTest.testV_1FGGUOO_1	146	AM/CM CompletionParser.
3	CompletionParserTest.testZ_1FGPF3D_1	146	consumeClassHeaderName()
4	DietCompletionTest.test16	146	LC(...codeassist.complete.CompletionParser,
5	DietCompletionTest.test17	146	...compiler.Parser.consumeClassHeaderName())
6	CompletionParserTest.testDB_1FHSLDR	164	CM AssistParser.popElement(int)
7	CompletionParserTest.testHB_1FHSLDR	164	CM CompletionParser.consumeToken(int)
			CM CompletionParser. createSingleAssistNameReference(char[],long)

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

Table 5.4: Details of test results for Eclipse jdt Compiler version pair 21 Jan 2003 – 22 Jan 2003. All tests in the same rectangle share the same failure-inducing changes.

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

Table 5.5: The comparison of the average numbers of failure-inducing changes of two definitions on Eclipse jdt compiler.

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% of all the atomic changes are affecting changes for each test; using *Crisp* we further isolate the failure-inducing changes to 1 to 4 changes.

5.2.3 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 a minimal 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 5.2 and 5.3 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 other **AF**s, **AC**s and **AM**s as prerequisites; thus, use of the pessimistic definition results in considering more changes as failure-inducing. Table 5.5 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 5, we notice a significant difference between the two definitions.

It is important to investigate whether or not each prerequisite for a failure-inducing change is ONLY a prerequisite for that change or is a prerequisite for other changes as well. If a prerequisite is shared with other **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 version pair 5 and found that about half of the prerequisites occurred solely as prerequisites of the failure-inducing changes; the other half 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.

Chapter 6

Heuristics for Locating Test Failure Causes

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. For large applications, the parts of an edit are inter-related in many ways. In some cases, the combination of some changes results in the failure of a specific test, making it more difficult and time-consuming to identify the failure causes.

In this chapter, we propose a heuristic that ranks the method changes affecting a failing test, indicating the likelihood that they were responsible for the failure. The basic idea of the heuristic is that if the programmer changes a method which has many ancestors and descendants in the test's call graph, and furthermore, if the caller and callee in the call graph are changed at the same time, these changes are more likely to introduce a failure. We also discuss other metrics on methods that may be used for heuristics.

We conducted a case study with our heuristic to rank all the method changes in *Eclipse jdt core* project, using the test suite from its *compiler tests* plug-in. To measure the effectiveness of our heuristic, we use *Crisp* [45] to determine the actual failure-inducing changes. Then we check how well we ranked the real failure-inducing changes among all the method changes in the affecting changes. Ideally, we would like to see the failure-inducing changes for a test always obtain the highest rankings. In the case study, for the tests whose failure is caused by one changed method, our heuristic ranks the failure-inducing change among the top 2 of all method changes for 67% such tests. For failures caused by combinations of the changes, our heuristic helps in half of the cases, in which the rankings we obtain are only one off from the ideal ranking.

We did the study in our change impact analysis framework, but our heuristic for ranking the changes can be used in a more general setting. Suppose a Java program fails unexpectedly, and the programmer can easily obtain a previous version of the program that works. He first can collect all the changes between the two versions, and build a call graph for the failed test in the edited version. Next, he can traverse the call graph and collect the properties for ranking the changed methods. Note that calculating the affecting changes for the failed program is not necessary; we can directly match the nodes in the call graph with the edit changes. Those method changes not related to the failed test won't appear in the test's call graph and will be ignored automatically.

6.1 Heuristics to Look for Failure Causes

6.1.1 An Informal Overview of the Approach

Change label	Changed Methods	# of Des	# of Anc	# of Callees	Callee chgd?	# of Callers	Caller chgd?	Score
2	SAcnt.crtTrans(int,long)	1	3	1	no	2	yes	6
9	Acnt.deposit(long)	2	1	1	yes	1	no	4
10	Acnt.withdraw(long)	2	1	1	yes	1	no	4
8	Acnt.<init>()	0	2	0	no	1	no	2
11	Acnt.curBallance()	0	1	0	no	1	no	1

Table 6.1: The properties and the scores of **CM** changes for test `Tests.testAcnt()` in the example program

We will use the example program of Figure 6.1 to illustrate how the heuristics work. The program of Figure 6.1(a) depicts a simple program comprised of classes `Trans`, `Acnt` and `SAcnt`, which represent *Transaction*, *Account* and *Savings Account* respectively. Figure 6.1(b) shows an edited version of the program, where the changes are shown underlined, labeled with the numbers of the corresponding atomic changes, shown in Figure 6.2. Associated with the program is a *JUnit* [32] test, `Tests.testAcnt`, which is shown in Figure 6.1(c). Note that it is assumed that the test will be used with both versions of the program.

Figure 6.2 shows the atomic changes that define the edit of the example program.

```
public class Trans {
  public static int DEPOSIT = 0;
  public static int WITHDRAW = 1;
  private Acnt acnt;
  private int type;
  private long amt;
  public Trans(Acnt acnt, int t, long amt) {
    this.acnt = acnt;
    this.type = t;
    this.amt = amt;
  }
}
public class Acnt {
  long balance = 0;
  List ts = new LinkedList();
  public void deposit(long amt) {
    crtTrans(Trans.DEPOSIT, amt);
    balance += amt;
  }
  public void withdraw(long amt) {
    crtTrans(Trans.WITHDRAW, amt);
    balance -= amt;
  }
  boolean crtTrans(int type, long amt) {
    ts.add(new Trans(this, type, amt));
    return true;
  }
  public long curBalance() {
    return balance;
  }
  public Acnt() { }
}
public class SAcnt extends Acnt {
}
```

(a)

```
public class Tests extends TestCase {
  public void testAcnt(){
    Acnt a = new SAcnt();
    a.deposit(100);
    a.withdraw(100);
    Assert.assertEquals(a.curBallance(),0);
  }
}
```

(c)

```
public class Trans {
  public static int DEPOSIT = 0;
  public static int WITHDRAW = 1;
  private Acnt acnt;
  private int type;
  private long amt;
  public Trans(Acnt acnt, int t,
  long amt) {
    this.acnt = acnt;
    this.type = t;
    this.amt = amt;
  }
}
public class Acnt {
  long balance = 0;
  List ts = new LinkedList();
  public void deposit(long amt) {
    if (crtTrans(Trans.DEPOSIT, amt))[9]
      balance += amt;
  }
  public void withdraw(long amt) {
    if (crtTrans(Trans.WITHDRAW, amt))[10]
      balance -= amt;
  }
  boolean crtTrans(int type, long amt) {
    ts.add(new Trans(this, type, amt));
    return true;
  }
  public long curBalance() {
    assert (balance >= 0);[11]
    return balance;
  }
  public Acnt() { no = ++index;[8] }
  private long no;[6]
  static private long index;[7]
}
public class SAcnt extends Acnt {
  boolean crtTrans(int type,
    long amt)[1,2,3,4] {
    if (type==Trans.WITHDRAW &&
      balance-amt<25)
      return false;
    ts.add(new Trans(this, type, amt));
    return true;
  }
  private double iRate;[5]
}
```

(b)

Figure 6.1: **(a)** Original version of example program. **(b)** Edited version of example program (underlining is used to show changed code fragments. **(c)** Tests associated with the example of (a) and (b).

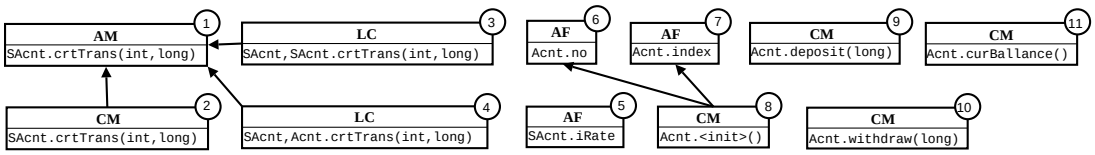


Figure 6.2: Atomic changes for the example program, with their inter-dependences

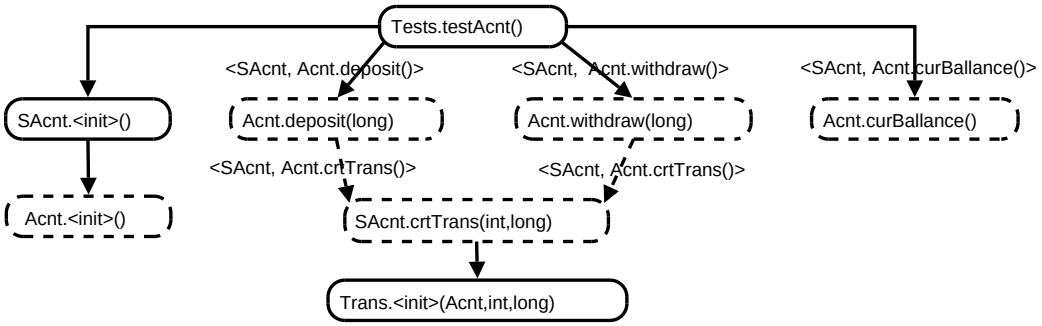


Figure 6.3: Call graph of the `Test.testAcnt()` executed on edited program in Figure 6.1, dashed boxes indicate changed methods, and dashed arrows indicate changed calling relationships between methods (lookup changes). Calls to the library methods are not shown.

An arrow from an atomic change A_1 to an atomic change A_2 indicates that A_1 is dependent on A_2. Recall that the *syntactic* dependences do not capture all *semantic* dependences between changes. For example, there isn't any syntactic dependence between changes to method `Acnt.withdraw(long)` (**CM** change 10) and the newly added method `SAcnt.crtTrans(int, long)`(**AM** change 1); however, these changes have a semantic dependence. Both changes affect the behavior of `testAcnt()`, and applying either of them alone will not affect the successful test result, but applying both changes to the original program causes the test to fail.

Test `testSAcnt()` fails in the edited program. Figure 6.3 shows the call graph for the test in the *edited* program. We use dashed boxes to indicate changed methods (**CM** changes), and dashed arrows indicate changed calling relationships between methods (**LC** changes). The set of atomic changes that affect a given test includes: (i) all changed methods (**CM** changes) that correspond to a node in the call graph in the edited program, (ii) lookup changes (**LC**) that correspond to an edge in the call graph, and (iii) their transitively prerequisite atomic changes.

For `testAcnt()`, category (i) includes changes 2, 8, 9, 10, 11 and category (ii)

includes change 4. From the dependence graph in Figure 6.2, it can be seen that atomic changes 2 and 4 require atomic change 1, and atomic change 8 requires atomic changes 6 and 7. Therefore, the affecting changes for `testAcnt()` are atomic changes 1, 2, 4, 6, 7, 8, 9, 10 and 11. The heuristic ranks the changes in category (i) when the test has failed.

In this section, we present a heuristic ranking of the **CM** changes in the affecting changes set, to indicate the likelihood they may have contributed to a test failure. Then we explain possible ways of exploring the changes given the ranking to quickly pinpoint the failure-inducing changes.

6.1.2 Heuristic

The heuristic is based on the hypothesis that those changed methods with large numbers of ancestors and/or descendants in the call graph are more likely to be failure-prone than other changed methods. For each node in the call graph of the test on the edited program that corresponds to a **CM**, a score is assigned based on some properties we collect. Then we sort all the **CM**s in *descending* order according to the score obtained from the call graph, so that the change with the highest score gets the top ranking (i.e., number 1 is the highest rank). The higher its rank, the more likely it is that the **CM** may have contributed to the test failure.

We collect several properties of a changed method, calculate a score from these properties, and use the score as our heuristic. If a given node in the call graph of the failed test is changed by the edit, we record the following information: (a) the number of descendants (including all callees), (b) the number of ancestors (including all callers), (c) the number of callees, (d) whether the callees include any node corresponding to a changed method, and (e) the number of callers, (f) whether the callers include any node corresponding to a changed method.

The score for each changed method is then calculated. First, the numbers of descendants (a) and ancestors (b) are counted. Then we check property (d), whether the callees include any node corresponding to a changed method. If so, the number of the callees (c) is included in the score; otherwise, the number is ignored. The process for

properties (e) and (f) is similar.

If two methods with the caller-callee relationship are changed at the same time, they may interact with each other and are more likely to be failure-prone, especially when these changes were made by different developers. In such cases, we want to assign a higher weight to these changed methods in the call graph. Suppose a method is changed by the edit, and one or more of its callees are changed at the same time. We can count the number of its *changed* callees and include the number in the score. However, compared to the total number of descendants/ancestors, this number is usually much smaller and has no effects on the final score for ranking. Since the changed callee(s) may have side-effects on the sibling callgraph nodes, and thus introduce more complexity, we chose to count the numbers of *all* the callees of the changed method whenever some of the callees are changed. This number is usually bigger than the number of *changed* callees and it does help balance the final ranking scores and avoids the use of sophisticated weighting functions for the changed callees. Similar processing is applied to the callers of the changed method. Note that the callees/callers of the changed method are counted twice by our heuristic if some of the callees/callers are also changed.

Call graphs may include cycles, so that the ancestors and descendants are not always disjoint sets. In determining the ranking of a changed method, we simply remove from the set of descendants, the method nodes that already appear in the set of ancestors. Thus, a specific method node is only counted once either as an ancestor or a descendant. Similar processing is done for direct callers and callees in such situations.

For the example program shown in Figure 6.1, there are 5 **CM**s in the affecting changes set of the test `Tests.testAcnt()`. Table 6.1 shows the properties and the scores of the 5 **CM**s, ordered by their rankings. The first column lists the change labels from Figure 6.2 for the changed methods shown in the second column. The third to the eighth columns show the properties (a) to (f) discussed above. The last column is the score obtained for the method, which is used for ranking.

In the above example, changed method `SAcnt.crtTrans(int, long)` has 1 descendant and 3 ancestors. From Figure 6.3, we notice that its callee does not change but its 2 callers are both changed methods, so the score for the method is 1 descendant plus 3

ancestors and plus 0 callee and plus 2 callers, which is 6 according our heuristic.

6.1.3 Explore the changes

We use *Crisp* [45] to determine the actual failure-inducing changes. Given the change rankings obtained in Section 6.1.2, there are several ways to explore the changes to pinpoint the failure-inducing changes by using *Crisp*.

Accumulative Sequential Exploring. Since we believe that the ranks of the **CM**s indicate the likelihood they may contribute to the failure of the test, the intuitive exploration strategy is to apply the **CM**s one by one in the ranking order (i.e., accumulatively) to the original program until the failure occurs. Then we undo all the changes applied, and re-apply the last **CM** change (i.e. the m^{th} **CM** change) to the original program and check whether the failure still occurs. If the test still fails, then the m^{th} **CM** change is the failure-inducing change. Otherwise, we know that the failure is caused by combinations of several **CM**s, and the m^{th} **CM** change is one of them. In such a case, we will treat the intermediate program, formed by the original program plus the m^{th} **CM** change, as the original program in the next run, remove this **CM** change from the affecting changes set, and then repeat the above procedure. In each run, we find one more element contributing to the failure of the test and the process will finally terminate because the affecting changes set is finite.

Another possible sequential exploring option is to apply the changes to the original program one by one, undoing the last change before applying the next one. But it is not effective when the failure is caused by the combination of several method changes, so we choose the accumulative sequential exploring in our experiment.

Divide-and-Conquer Exploring. This is similar to the idea used in delta-debugging [65], but the division of the changes is not random. We evenly divide the changes into two subsets in ranking order, with the higher ranked **CM**s in the first subset, and the lower ranked **CM**s in the second subset. We always apply the first subset to the original program first, and check whether the resulting intermediate program results in test failure. If the test fails, then we repeat the procedure on the first subset; otherwise, we try the second subset.

This procedure has three possible outcomes: 1) we find one **CM** change that results in the test failure; 2) several combined changes cause failure, and they are in the same subset (In this case, we need to confirm that this is the minimum group of failure-inducing changes); 3) several combined changes cause failure, but they are in different subsets, which means applying the combination of two subsets makes the test fail, but neither of the subsets alone causes the failure. In such a case, we will extend the first subset by adding the changes from the beginning of the second subset, one change a time, until the test fails. Finally, we use accumulative sequential exploring on the extended subset to find the failure-inducing changes.

After finding a group of failure-inducing changes F, the programmer still needs to check whether these are the only changes that cause the test to fail. He can apply the complement of these changes to the original program and run the test on the resulting intermediate program[1]. If the test still fails, he should continue the above process on the remaining changes and find all the failure-inducing changes. If the test succeeds, then the complement set contains no failure-inducing changes. Note that applying only part of the complement set may cause test failure because of possible semantic dependences between changes in the complement set.

	Accumulative Sequential		Divide-and-Conquer	
Steps	Changes	Outcome	Changes	Outcome
1	2	P	2, 9, 10	F
2	2, 9	P	2, 9	P
3	2, 9, 10	F	10	P
4	10	P	2, 10	F
5	2, 10	F	6, 7, 8, 9, 11	P
6	6, 7, 8, 9, 11	P		

Table 6.2: The changes applied in each step using the two exploration strategies and the outcomes of the intermediate programs in each step. Undo and re-apply steps are not shown in the table.

Example. Table 6.2 shows the atomic changes examined in each step of the two exploration strategies and the outcomes of the intermediate programs for the example in

[1]The complement set of F includes the all the affecting changes that are not in F and their prerequisites.

Figure 6.1. The undo and re-apply steps are not shown in the table. Using accumulative sequential exploring, we apply the **CM** changes 2, 9 and 10, and then observe the test failure. Then, we undo all the changes, and then re-apply change 10, but the test doesn't fail. Now we know that the failure is caused by the combination of change 10 and one or both of the other two changes (changes 2 and 9). Next we apply atomic change 2 accumulatively on atomic change 10 and the test fails. The final step is to undo all the changes applied and apply their complement to the original program and confirm that the complement does not result in a failure.

The divide-and-conquer strategy works in a different way. We apply the first subset of the **CM** changes (changes 2, 9 and 10) and observe the test failure. Then, we undo the changes, further subdivide the current subset into two parts, and re-apply the first part (atomic changes 2 and 9), and observe that the test succeeds. Then we try the second part of the changes (change 10), and the test still succeeds. So we must extend the first part by adding a change from the second part. Since we already tried the combination of atomic changes 2, 9, 10 in the first step, we use the accumulative sequential exploring strategy on the extended subset. Finally, we locate atomic changes 2 and 10 as the failure-inducing changes. Again we need to check the complement to make sure that we have found all the failure-inducing changes.

6.2 Eclipse jdt Case Study

We performed our case study on an *Eclipse* plug-in project, `org.eclipse.jdt.core`, which has several associated test plug-ins. Several test plug-ins are associated with `jdt core` and we chose the `org.eclipse.jdt.core.tests.compiler` plug-in to do the case study because of its availability with the development history of `jdt.core`. In addition, the tests in this plug-in are all *JUnit* tests.

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 `jdt.core.tests.compiler` plug-in, attempting to find failing tests.

We succeeded in finding 3 version pairs with at least 6 affecting **CM** changes, that

Index	Version pairs	# of Atomic Changes	# of Failed Tests	Avg. # of Affecting Changes	Avg. # of **CM**s
1	1/21/2003–1/22/2003	724	7 (5, 2)	151	18
*2	3/31/2003–4/1/2003	371	1 (1, 0)	42	10
3	1/13/2004–1/14/2004	163	8 (8, 0)	74	42
4	2/17/2004–2/18/2004	2701	2 (2, 0)	63	23
5	3/4/2004–3/5/2004	68	240 (48, 192)	13	7
*6	4/10/2004–4/11/2004	146	46 (46, 0)	37	13
7	6/8/2004–6/9/2004	116	2 (2, 0)	12	9
8	6/9/2004–6/10/2004	719	2 (0, 2)	8	8
9	7/9/2004–7/10/2004	103	1 (1, 0)	16	8
10	7/21/2004–7/22/2004	163	7 (7, 0)	16	9
*11	11/16/2004–11/17/2004	465	1 (1, 0)	22	9
12	11/18/2004–11/19/2004	1784	2 (2, 0)	36	15
13	11/19/2004–11/20/2004	325	2 (2, 0)	153	32
14	11/29/2004–11/30/2004	234	1 (0, 1)	11	9

Table 6.3: The summary of the version pairs of `jdt.core` in *Eclipse* used in our case study

have successful test results in the original version and failing tests in the next nightly build (edited) version, which are indicated by a star in the index column in Table 6.3. In order to simulate the development life cycle of the program, 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. Using this strategy, we found more version pairs where some tests were successful in version n but failed in version $n + 1$.

Then we ran *Chianti* to generate the atomic changes for each version pair, and to calculate the affecting changes for each failed test. Some of the tests have small sets of affecting changes and thus it is easy to locate the causes of failures. So for our experiments, we ignored those tests with less than 6 **CM** changes in the affecting changes set.

Finally we obtained 14 version pairs on which to conduct our case study, which is summarized in Table 6.3. For the versions under consideration, the project increased in size from 148K to 204K uncommented lines of source code and the number of JUnit tests grew from 1069 to 5224 [2]. The second column in Table 6.3 lists the version pairs of

[2]The total number of JUnit tests depends on the Java version used in *Eclipse*; we used JDK1.4 in the experiment.

the project `jdt.core` represented by their dates. The third column lists the number of atomic changes for each version pair. The fourth column is the number of failed tests. Some tests have a single failure-inducing change, which means that by applying one **CM** to the original program, we generate the expected failure for the test. Some test failures required multiple method changes. In parentheses in this column, we list a pair of numbers, representing the number of tests with a single failure-inducing method change, and the number of tests with multiple failure-inducing method changes, respectively. The last two columns show the average number of affecting changes for each failed test per version and the average number of **CM** changes among these affecting changes.

6.2.1 Data Analysis

Single failure-inducing change

Among the 14 version pairs, 10 of the version pairs have tests with a single failure-inducing method change. In addition, for version pair 1, there are 7 failed tests in total, 5 of which have a single failure-inducing method change, while the other two tests were failed by multiple combined method changes, and the failure-inducing changes for these two groups do not intersect with each other.

Figure 6.4 shows the number of failed tests with at least 6 **CM** changes for these 11 version pairs. Each column represents the average number of affecting **CM** changes over the failed tests. Note that for version pair 1, the numbers in the figure are different from the numbers shown in Table 6.3, which include those tests failed by multiple combined method changes. The lower bar in each column shows the average rank on the failure-inducing changes (note the log scale on the numbers for the lefthand side axis). Ideally, we would always rank the failure-inducing change as number 1, the top rank. Generally, the higher the ranking, the less work to locate the failure-inducing changes. The diamond-shaped dots connected by a line show the number (in the frame box) of failed tests for each version pair (see the righthand side axis).

Versions details. Figure 6.4 shows that in 4 of 11 version pairs, we correctly ranked the failure-inducing change as number 1 (i.e., version pair 2, 7, 9 and 11); in 3 of 11

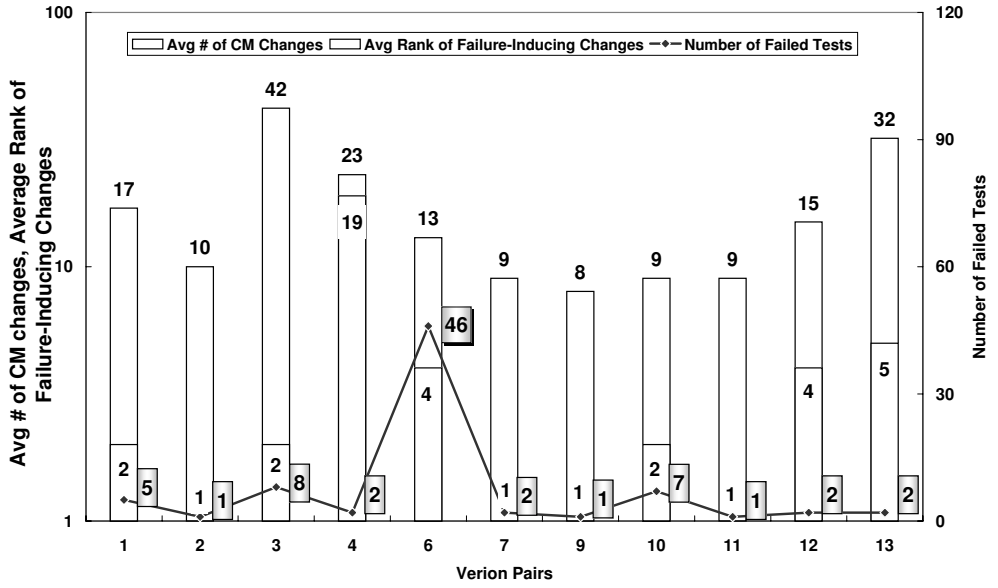


Figure 6.4: The number of failed tests (with at least 6 affecting **CM** changes) versus the average ranks of the failure-inducing changes (note the log scale).

version pairs, the failure-inducing change was ranked number 2 on average (i.e., version pair 1, 3 and 10). For version pair 3, there are 74 affecting changes for the tests, and 42 of them are method changes; our heuristic successfully ranked the failure-inducing change as number 2.

In version pair 12, we ranked the failure-inducing change as number 4 of all 15 method changes. Note that in version pairs 6, 12 and 13, we ranked the failure-inducing change in the top $log_2 n$ rank, where n is the number of **CM** changes in the affecting changes set.

In version pair 4, the failure-inducing change was ranked as number 19 among 23 changes. After checking the details of the test, we found that this failure is caused by a change to a static field initializer, thus resulting in a change to the class initializer method $\langle clinit \rangle()$. Since this method is called only when the class is first loaded into the system, it has no parent in the call graph and usually has few callees, so its score is always low. Our heuristic doesn't work well here. In fact, other research work in fault localization [31, 3] also confirmed that it's difficult to diagnose faults located in the initialization code, even for small programs.

Variations. All the ranks we show above are averages across all the failing tests

Index	Best Rank	Worst Rank	Median Rank
1	1	4	1
3	2	2	2
4	19	19	19
6	2	4	4
7	1	1	1
10	1	2	2
12	4	4	4
13	5	5	5

Table 6.4: The variance of the ranks of failure-inducing changes for the tests in each version pair.

per version. For those version pairs with multiple failing tests, we are also interested in the variations between the ranks of the failure-inducing changes. Table 6.4 shows the best rank, the worst rank and the median rank of the failure-inducing change for each version pair with more than one failing test. In 5 of the version pairs (version pair 3, 4, 7, 12 and 13), we assign the failure-inducing changes the same ranking for different failing tests. Version pair 1 has the biggest variation. There are 5 failing tests with a single failure-inducing change; for 4 of them, our heuristic ranks the failure-inducing change as number 1 out of 17 **CM** changes, and in one test, we rank the failure-inducing change as number 4 out of 15 **CM** changes.

For version pair 6, there are 46 failed tests in the edited version; all these tests share the same failure-inducing change. In 12 of the failed tests, we ranked the failure-inducing method change as number 2, and for the other 34 of the failed tests, the failure-inducing change was ranked as number 4. For all the failed tests, two non-failure-inducing **CM** changes obtain the same score as or very close to (i.e., the score difference less than 2) the failure-inducing **CM** change. These three method changes are always ranked between 2 and 4. We checked other attributes of these three methods, and found that the failure-inducing change method has more siblings than the other two methods in the call graph (30 vs. 8,9). This may be a useful hint to prioritize the changes when their scores are the same or very close.

Common failure-inducing changes in failing tests. Another interesting question is whether different failing tests in a version pair share the same failure-inducing

changes. For the version pairs shown in Figure 6.4, all the failing tests, except one in version pair 1, share the same failure-inducing change with the other tests in the same version pair. In version pair 1, one test with the failure-inducing change ranked as 4 has a different failure-inducing change than the other 4 failing tests.

This implies that often we can start with one failing test, search for the failure-inducing changes for this test, and after locating the them, run all the other failing tests and check whether only the same set of changes make all fail. In this case, we can avoid costly call graph collection for all the failed tests, thus saving programmer time and effort in debugging.

Given our observations above, we hypothesize that many failed tests in the same version pair share the same failure-inducing changes, and thus we don't need to locate the failure-inducing changes for these tests one by one. Therefore, instead of calculating average rank per test, we developed a more realistic measure of how rank helps in the debugging process. We represent each of these sets of tests with a *delegate test*; there are 12 delegate tests for the 11 version pairs we discussed above, where version pair 1 includes 2 delegate tests and each of the other version pairs include 1 delegate test. Table 6.5 shows the distribution of the ranks of the failure-inducing changes for the delegate tests. In 6 of the cases, the failure-inducing change was ideally ranked as number 1 out of 10 **CM** changes on average, and in 2 cases, the failure-inducing change was ranked as number 2 out of 28 **CM** changes on average. In summary, our heuristic ranked the failure-inducing change within the top 2 over all the **CM** changes for 67% (8 out of 12) of the delegate tests.

Avg # of changed methods	Rank of Failure-inducing Change	# of Delegate Tests
10	1	6
28	2	2
15	4	2
32	5	1
23	19	1

Table 6.5: The distribution of the ranks of the failure-inducing changes for the delegate tests

Multiple failure-inducing changes

In version pair 1, two tests `testHB_1FHSLDR` and `testDB_1FHSLDR` defined in class `parser.CompletionParserTest` failed in the edited version because of three combined **CM** changes. Test `testHB_1FHSLDR` has 21 affecting **CM** changes, and the failure-inducing changes are ranked as 2, 3, and 4, only one off from the ideal ranking. Similar to what we observed in cases with a single failure-inducing change, these two tests share the same failure-inducing changes set. We also confirmed that (i) any subset of the changes doesn't generate the expected failures for either test, and (ii) the complement of the changes doesn't generate failure for the tests, either. For `testDB_1FHSLDR`, the failure-inducing changes are ranked as 2,3, and 7 out of 21 affecting **CM** changes.

In version pair 8, two tests `test078` and `test077` defined in class `regression.Compliance_1_4` failed in the edited program. Both failures are caused by two combined **CM** changes, but the failure-inducing changes sets are not exactly the same; only one common **CM** change is shared between them. `test078`'s failure-inducing changes are ranked as 2 and 3 out of 7 affecting **CM** changes (one off from the ideal ranking) and `test077`'s failure-inducing changes are ranked as 2 and 6 out of 8 affecting **CM** changes; our heuristic doesn't help as much here.

In summary, for the failures caused by combinations of changes, our heuristic helps in half of the test cases; the rankings we obtain are only one off from the ideal rankings.

Combination of failure-inducing changes

The failure of one test in version pair 14, can be caused by each of three **CM** changes separately. Our heuristic ranks the failure-inducing changes as 2, 4, and 5 out of 9 **CM** changes respectively.

Version pair 5 has 240 failed tests in the edited program, and we divide them into three groups by their failure-inducing changes. Twelve of them were failed by the change to method `jdt.internal.compiler.problem.ProblemReporter.computeSeverity(int)`, which is ideally ranked as number 1 out of 7 **CM** changes. Thirty-six of them were failed by

the change to the initialization code `jdt.internal.compiler.parser.JavadocParser.<init>(jdt.internal.compiler.parser.Parser)`, which is ranked as number 4 out of 6 **CM** changes. All the other 192 failed tests share the same affecting changes set (7 of them are **CM** changes), and were failed by each of these method changes separately.

Figure 6.5 shows a scatter plot of the rankings for the failed tests in version pair 5. The X axis represents the rank of the failure-inducing change `ProblemReporter.computeSeverity(int)`, and Y axis represents the rank of the failure-inducing change `JavadocParser.<init>(Parser)`. The size of each bubble reflects the number of same-valued pairs it represents, which appears explicitly as a number next to the bubble. Bubbles close to the (1,1) point are the most desirable values, that is the failure-inducing changes were ranked close to 1. The bubbles on the X axis and Y axis represent those tests with a single-failure-inducing change. For example, the bubble on the X axis represents the 12 tests failed by the method change `ProblemReporter.computeSeverity(int)`, which is ranked as number 1.

For those 192 tests failed by either method separately, these two methods obtain different rankings in different tests. The biggest bubble in the graph represents 187 of these, the change to method `ProblemReporter.computeSeverity(int)` was ranked as 1, and the change to method `JavadocParser.<init>(Parser)` was ranked as 5. The ranks of the failure-inducing changes for the other 5 tests are worse.

In summary, for this version pair, the failure-inducing change `ProblemReporter.computerSeverity(int)` was easy to locate, since in most cases, our heuristic ranks this method change ideally, as number 1. However, the change to the initialization code `JavadocParser.<init>(Parser)` is hard to locate (i.e., the rankings for this method change are always very low). This is consistent with what we observed in version pair 4 and also was discussed by other research work in fault localization [31, 3]

6.2.2 Comparison to Other Heuristics

Our previous work [12, 45] presented an exploration strategy to locate the failure-inducing changes. As mentioned in Chapter 4, syntactic dependences are calculated between atomic changes to ensure the compilability of the intermediate programs. If

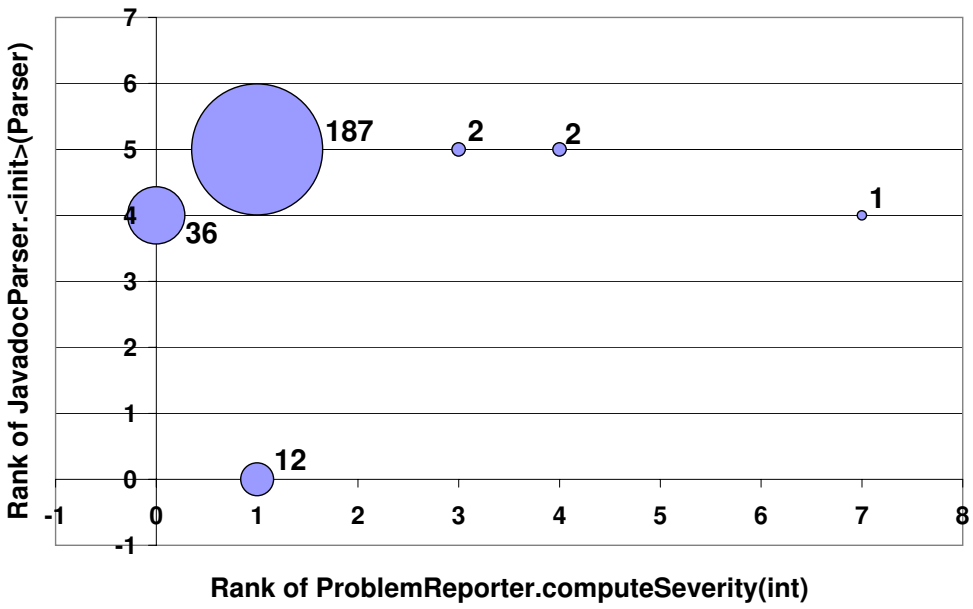


Figure 6.5: Scatter plot for ranks of two failure-inducing changes for version pair 5.

a programmer wants to apply an atomic change to the original program, all of its prerequisite changes must also be applied. The basic idea of this exploration strategy is to define the complexity of an atomic change according to the number of its prerequisite changes, and always to apply the simple atomic changes first, then the more complicated changes (i.e., fewest prerequisites first).

Figure 6.6 shows the comparison of rankings of the failure-inducing changes by two heuristics applied to each set of tests represented by a delegate test. We refer to the heuristic presented in this paper as CS heuristic, since the ranking depends on the calling structure of the method. We use PR to represent the exploration strategy presented in [12, 45] since it is defined using the number of prerequisite changes. This figure only shows the rank comparison for the tests with a single failure-inducing change. Because version pair 1 includes two groups of tests with different failure-inducing changes, columns 1 and 1* represent these two groups, respectively. For each heuristic, Figure 6.6 shows the average ranking of the failure-inducing change, and marks the best and worst rankings it can achieve.

As mentioned before, many tests share the same failure-inducing change, which might be assigned different scores in distinct tests because of differences in calling

structures. In our experiment, the calling structures for the failed tests in the same version pair are similar to each other, which explains why for the CS heuristic, there is very seldom a difference between the best ranking and the worst ranking. The PR strategy ranks the **CM** changes based on their number of prerequisites. However, many **CM** changes share the same dependence complexity and thus cannot be distinguished by the number of prerequisites. The figure shows that in 8 of the 12 pairs, the failure-inducing change can't be distinguished by PR from other affecting **CM** changes and they share the same ranking. For example, in version pair 3, the failure-inducing change has 1 prerequisite and is ranked as number 1, but 17 other affecting **CM** changes also have 1 prerequisite and thus the same rank as the failure-inducing change. In such cases, the programmer has to try those 18 **CM** changes randomly until the test fails. In 10 of the 12 pairs, the CS ranking works better than the PR ranking on average. Even for the best ranking the PR can achieve, CS performs no worse than PR in 9 of the 12 pairs.

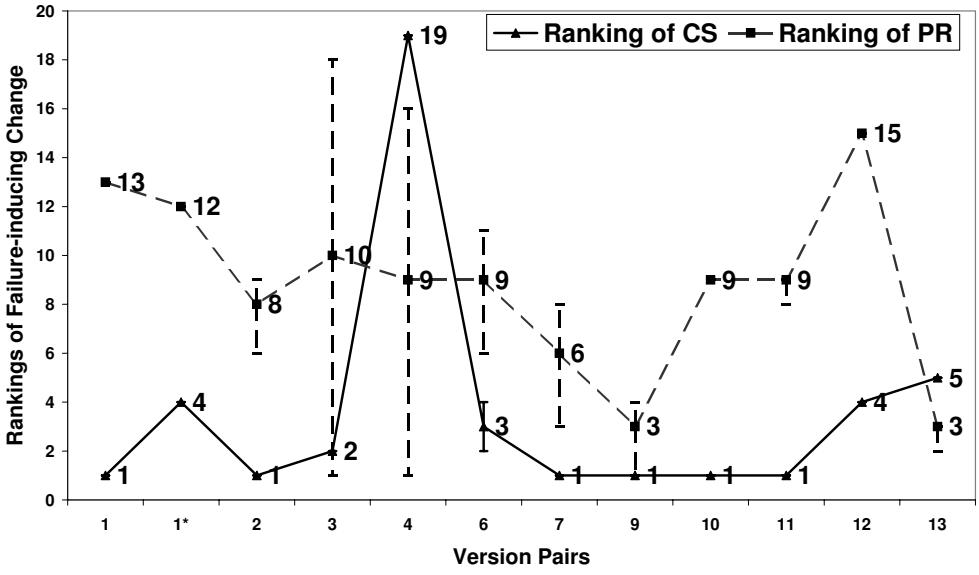


Figure 6.6: Comparison of CS and PR heuristics

Other factors affecting the ranking? Our proposed heuristic is very intuitive. We also tried to assign weights to other properties of the changed method to rank the **CM** changes. For example, we considered the number of siblings of the changed method, whether the siblings have been changed or not, the number of fields accessed in each

method, the number of classes/interfaces referred to by each method, the number of changed callers, callees, and siblings, the numbers of the different types of prerequisites (e.g., class, field and method prerequisites) of the changed method. However, for the data in our case study, those properties had no significant influence on the rankings. Therefore we abandoned the use of these extra properties and kept the algorithm as simple as possible.

We also are interested in the sensitivity of the heuristic to the current choice of properties. We need to do more experiments to further explore these issues.

6.2.3 Assessment

We performed our case study on 14 version pairs of a real Java project, and observed a good combination of single and multiple failure-inducing changes. For the tests with a single failure-inducing change, our heuristic successfully ranked the failure-inducing change as number 1 for 50% of the delegate tests. For most cases, we ranked the failure-inducing change in top $log_2 n$, where n is the number of method changes in the affecting changes set of the test. However, our heuristic didn't work at all when the failure are caused by the changes to the initializer code.

Generally, if a test failure is caused by several combined changes, there is no easy way to pinpoint these changes. Our approach helps in several ways in this situation. First, the change impact analysis from *Chianti* filters out the changes not related to the failed test. Second, the dependences between atomic changes allow the programmer to focus only on method changes that may cause the failure. Third, when the initializer code is not involved in the failure of tests, the study shows that our heuristic helps in half of the cases, in which the rankings we obtain are only one off from the ideal rankings. This saves a programmer considerable time compared to trying all possible combinations of the affecting changes.

When a programmer experiences a test failure, she has no idea whether it is caused by a single change or some change combination. Since our heuristic generates useful information in most cases, we recommend the programmer use the accumulative sequential exploring of the method changes according to our rankings at first. If, after

applying $log_2 n$ method changes, she still can not generate the expected test failures, which is a hint that our ranking algorithm doesn't fit in such a case, she should consider continuing with the divide-and-conquer exploring strategy to accelerate the process.

6.2.4 Limitations

In our heuristics, we only consider the **CM** changes, and ignore the **LC** changes. In most cases, **LC** is a result of overriding a method, and thus the target of the dynamic dispatch resolves to the newly added method. Since this implies a newly implemented method, there is always an associated **CM** change. However, the **LC** change can also be caused by deleting a method, in which case we may not generate this failure even after we apply all the **CM** changes (i.e., if the **LC** change is the only failure-inducing change). If we examine the complement of the applied changes, we finally could locate the **LC** change, but not in an effective way. In our case study, this never happened, because programmers usually delete methods very carefully, since they suspect that a method may be used somewhere.

The other limitation is that we map all changes to method-level changes, and only rank the **CM** changes. In some cases, if the programmer changes many static field initializers, we will map them all to one class initializer change. In version pair 4, we locate the $CM(\langle clinit \rangle())$ as the failure inducing change; however, it represents more than one static field initializer change. We need to continue applying changes in a finer grained manner and finally locate the specific field changes that actually make the test fail.

6.2.5 Machine Learning Algorithms

We tried several machine learning algorithms to help locate the failure-inducing changes; however, they did not perform well in our case study for several reasons. Generally, the more data available for training, the better the results that the machine learning algorithms produce. We have only 14 version pairs in the study. The large number of failing tests (322) doesn't help too much because for each version pair, many of them share the same affecting changes set and the same failure-inducing changes.

In addition, to use a supervised machine-learning algorithm, which fits in our study, the accuracy of the learned function depends strongly on how the input object is represented. In our case, the difficulty came from the representation of the ranking; a binary value was insufficient to sort the **CM** changes so that suspicious changes receive higher rankings. We will gather more data and try other machine learning algorithms in the future.

Chapter 7

Related Work

We distinguish five broad categories of related work in the community: (i) change impact analysis techniques, (ii) regression test selection techniques, (iii) fault localization, (iv) techniques for avoiding recompilation, and (v) techniques for controlling the way changes are made and understanding the changes.

7.1 Change Impact Analysis Techniques

Previous research in change impact analysis has ranged from approaches relying completely on static information, including the analyses in [5, 36, 33], to approaches that only utilize dynamic information, such as [35]. There also are some methods [40] that use a combination of static and dynamic information. The method described in this paper is a combined approach, in that it uses (i) static analysis for finding the set of atomic changes comprising a program edit and (ii) dynamic call graphs to find the affected tests and their affecting changes.

All previous impact analysis focuses on finding *constructs of the program potentially affected by code changes*. In contrast, our change impact analysis aims to find *a subset of the changes that impact a test whose behavior has (potentially) changed*. First, we will discuss the previous static techniques, and then address the combined and dynamic approaches.

An early form of change impact analysis used reachability on a call graph to measure impact. This technique[1] was presented by Bohner and Arnold [5] as "intuitively appealing" and "a starting point" for implementing change impact analysis tools. However,

[1] This is only one of the static change impact analyses discussed.

applying the Bohner-Arnold technique is not only imprecise but also unsound, because, by tracking only methods downstream from a changed method, it disregards callers of that changed method that can also be affected.

Kung *et al.* [33] described various sorts of relationships between classes in an object relation diagram (i.e., ORD), classified types of changes that can occur in an object-oriented program, and presented a technique for determining change impact using the transitive closure of these relationships. Some of our atomic change types partially overlap with their class changes and class library changes.

More recently, Tonella's impact analysis [59] determines if the computation performed on a variable x affects the computation on another variable y using a number of straightforward queries on a concept lattice that models the inclusion relationships between a program's decomposition (static) slices [25]. Tonella reports some metrics of the computed lattices, but gives no assessment of the usefulness of his techniques.

A number of tools in the Year 2000 analysis domain [19, 44] use type inference to determine the impact of a restricted set of changes (e.g., expanding the size of a date field) and perform them if they can be shown to be semantics-preserving.

Thione *et al.* [56, 55] wish to find possible semantic interference introduced by concurrent programmer insertions, deletions or modifications to code maintained with a version control system. In this work, a semantic interference is characterized as a change that breaks a def-use relation. Their unit of program change is a *delta* provided by the version control system, with no notion of subdividing this delta into smaller units, such as our atomic changes. Their analysis, which uses program slicing, is performed at the statement level, not at the method level as in *Chianti*. No empirical experience with the algorithm is given.

The *CoverageImpact* change impact analysis technique by Orso *et al.* [40] uses a combined methodology, by correlating a forward static slice [58] with respect to a changed program entity (i.e., a basic block or method) with execution data obtained from instrumented applications. Each program entity change is thusly associated with a set of possibly affected program entities. Finally, these sets are unioned to form the full change impact set corresponding to the program edit.

There are a number of important differences between our work and that by Orso *et al.*. First, we differ in the goals of the analysis. The method of Orso *et al.* [40] is focused on finding those program entities that are possibly affected by a program edit. In contrast, our method is focused on finding those changes that caused the behavioral differences in a test whose behavior has changed. Second, the granularity of change expressed in their technique is a program entity, which can vary from a basic block to an entire method. In contrast, we use a richer domain of changes more familiar to the programmer, by taking a program edit and decomposing it into interdependent, atomic changes identified with the source code (e.g., add a class, delete a method, add a field). Third, their technique is aimed at deployed code, in that they are interested in obtaining user patterns of program execution. In contrast, our techniques are intended for use during the earlier stages of software development, to give developers immediate feedback on changes they make.

Law and Rothermel [35] present *PathImpact*, a dynamic impact analysis that is based on whole-path profiling [34]. In this approach, if a procedure p is changed, any procedure that is called after p, as well as any procedure that is on the call stack after p returns, is included in the set of potentially impacted procedures. This technique combines the use of a forward static slice [58] 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. Although our analysis differs from that of Law and Rothermel in its goals (i.e., finding affected program entities versus finding changes affecting tests), both analyses use the same method-level granularity to describe change impact.

A recent empirical comparison [41] of the dynamic impact analysis *CoverageImpact* by Orso *et al.* [40] and *PathImpact* by Law and Rothermel [35] revealed that the latter computes more precise impact sets than the former in many cases, but uses considerably (7 to 30 times) more space to store execution data. Based on the reported performance results, the practicality of *PathImpact* on programs that generate large execution traces seems doubtful, whereas *CoverageImpact* [41] does appear to be practical, although it can be significantly less precise. Another outcome of the study is that the relative

difference in precision between the two techniques varies considerably across (versions of) programs, and also depends strongly on the locations of the changes.

Rajlich *et al.* [43, 29, 6] 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 [28] proposed a classification of runtime software changes to help programmers understand such changes. This work addresses a different change problem than our research.

7.2 Regression Test Selection

Selective regression testing[2] aims at reducing the number of regression tests that must be executed after a software change [50, 42]. These techniques typically determine the entities in user code that are covered by a given test, and correlate these against those that have undergone modification, to determine a minimal set of tests that are affected.

Several notions of coverage have been used. For example, *TestTube* [10] uses a notion of module-level coverage, and *DejaVu* [50] uses a notion of statement-level coverage. The emphasis in this work is mostly on reducing the cost of running regression tests, whereas our interest is primarily in assisting programmers with understanding the impact of program edits.

Bates and Horwitz [2] and Binkley [4] proposed fine-grained notions of program coverage based on program dependence graphs and program slices, with the goal of providing assistance with understanding the effects of program changes. In comparison

[2] We use the term broadly here to indicate any methodology that tries to reduce the time needed for regression testing after a program change, without missing any test that may be affected by that change.

to our work, this work uses more costly static analysis based on (interprocedural) program slicing and considers program changes at a lower-level of granularity, (e.g., changes in individual program statements).

Our technique for change impact analysis uses affected tests to indicate to the user the functionality that has been affected by a program edit. Our analysis determines a subset of those tests associated with a program which need to be rerun, but it does so in a very different manner than previous selective regression testing approaches, because the set of affected tests is determined without needing information about test execution on both versions of the program.

Rothermel and Harrold [50] present a regression test selection technique that relies on a simultaneous traversal of two program representations (control flow graphs (CFGs) in [50]) to identify those program entities (edges in [50]) that represent differences in program behavior. The technique then selects any modification-traversing test that is traversing at least one such "dangerous" entity. This regression test selection technique is *safe* in the sense that any test that may expose faults is guaranteed to be selected.

Harrold *et al.* [30] present a safe regression test selection technique for Java that is an adaptation of the technique of Rothermel and Harrold [50]. In this work, Java Interclass Graphs (JIGs) are used instead of control-flow graphs. JIGs extend CFGs in several respects: Type and class hierarchy information is encoded in the names of declaration nodes, a model of external (unanalyzed) code is used for incomplete applications, calling relationships between methods are modeled using Class Hierarchy Analysis, and additional nodes and edges are used for the modeling of exception handling constructs.

Unlike regression test selection techniques such as [50, 30], the method presented in this thesis does not rely on a simultaneous traversal of two representations of the program to find semantic differences. Instead, we determine affected tests by first deriving from a source code edit a set of atomic changes, and then correlating those changes with the nodes and edges in the call graphs for the tests in the original version of the program. Investigating the cost/precision trade-offs between these two approaches for finding tests that are affected by a set of changes is a topic for further research.

In the work by Elbaum *et al.* [20], a large suite of regression tests is assumed to

be available, and the objective is to *select* a subset of tests that meets certain (e.g., coverage) criteria, as well as an order in which to run these tests that maximizes the rate of fault detection. The difference between two versions is used to determine the selection of tests, but unlike our work, the techniques are to a large extent heuristics-based, and may result in missing tests that expose faults.

The change impact analysis of [40] can be used to provide a method for selecting a subset of regression tests to be rerun. First, all the tests that execute the changed program entities are selected. Then, there is a check if the selected tests are *adequate* for those program changes. Intuitively, an adequate test set T implies that every relationship between a program entity change and a corresponding affected entity is tested by a test in T. In their approach, they can determine which affected entities are not tested (if any). According to the authors, this is not a safe selective regression testing technique, but it can be used by developers, for example, to prioritize test cases and for test suite augmentation.

7.3 Fault Localization

7.3.1 Delta Debugging

In the work on *delta debugging*, the reason for a program failure is identified as a set of differences between versions [65], inputs [67], thread schedules [14], or program states [66, 15] that distinguish a successful program execution from a failing one. A set of failure-inducing differences is determined by repeatedly applying different subsets of the changes to the original program, and observing the outcome of executing the resulting intermediate programs. By correlating the outcome of each execution (*pass, fail*, or *inconsistent*), with the set of changes applied, one can narrow down the set of changes responsible for the failure using efficient binary-search techniques.

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. This is extremely useful when the re-execution of the regression test suites is costly. Second, delta debugging builds the intermediate versions by only using the structural differences between successful 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 semi-automatic, requiring programmer selection of the changes to be added. The two approaches may complement each other. In principle, the use of a rich model of changes with interdependences could improve the efficiency of delta debugging by reducing the number of intermediate programs that are constructed/executed.

7.3.2 Program Slicing

Program slicing [58] 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* [37] 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.* [16] 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% and retain the failure-inducing statement in 80% of the cases.

Bunus and Fritzson [7] suggest a semi-automatic 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.* [27] uses delta debugging to *simplify* or *isolate* inputs that are failure-inducing, and then uses 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.

7.3.3 Other Techniques for Fault Localization

Stoerzer *et al.* [54] presented an approach for change classification that helps programmers identify the changes responsible for test failures. It proposed several change classifiers that associate the colors *Red*, *Yellow*, or *Green* with changes, according to the likelihood that they were responsible for test failures. The major difference between our heuristic and the change classification approach is that the change classifiers require obtaining all the affecting changes for all the affected tests in a test suite, while our heuristic only requires affecting changes for the failed test. For the project in our case study, there are thousands of tests in the test suite; thus, the heuristic approach represents a time saving over change classification.

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 inter-unit dependences that can be supplied by the programmer (i.e., as in the UNIX *make* [23]) or based on derived syntactic or semantic relationships. These dependences, describing clients of changed program constructs, are *incomparable* to the dependences used in *Crisp*, which 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 [9, 39]).

The earliest work was *smart recompilation* by Tichy [57, 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.* [1] later compared several smart recompilation approaches empirically to quantify their benefits on several Ada programs, finding a savings of approximately 50% of the recompilation effort. Burke and Torczon [8] 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 [17] 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.

7.5 Techniques for Controlling and Understanding the Changes

Palantir [52] is a tool that informs users of a configuration management system when other users access the same modules and potentially create direct conflicts.

Lucas *et al.* [53] describes *reuse contracts*, a formalism to encapsulate design decisions made when constructing an extensible class hierarchy. Problems in reuse are avoided by checking proposed changes for consistency with a specified set of possible operations on reuse contracts.

Xing *et al.* [61] presents *UMLDiff*, an algorithm for automatically detecting structural changes between the subsequent versions Java programs. The structural changes are reported in terms of additions, removals, moves, renamings of packages, classes, interfaces, fields and methods, as well as changes to their attributes. Then in their recent work [60, 63, 62, 64], they present some applications of *UMLDiff*. For example, they defined some queries in [62] to recognize the refactorings that occurred in the evolution history of a software system; all the refactorings and change patterns are detected as combinations of the basic change facts extracted by *UMLDiff*. Reference [60] shows how they use the results of *UMLDiff* to perform three analyses (phasic, gamma and optimal sequence matching analysis) to recognize a high-level abstraction of distinct evolutionary phases and to identify class clusters with similar evolution trajectories. They also propose a data-mining method for recovering hidden co-evolutions of system classes based on their *UMLDiff* algorithm.

The main difference between our change impact analysis and their *UMLDiff* algorithm is how the edit between subsequent versions is measured. *UMLDiff* measures the differences between two subsequent versions by structural changes and ignores any possible syntactic dependences between these changes. In our framework, change impact is measured by the affected tests and affecting changes for a given affected test. And furthermore, we recognize the dependences between changes for automatic construction of the intermediate programs to help programmers pinpoint the failure-inducing changes easily. Since *UMLDiff* algorithm ignores the dependences between changes, it can't be used for debugging purpose; instead, the application of their algorithm is for

understanding the evolution histories of a program. Part of our future works is to analyze our dependence graph of the atomic changes to recognize high-level design changes and help understand the evolution of a program.

Chapter 8

Summary and Future Work

Change impact analysis consists of a collection of techniques for determining the effects of source code modifications to improve programmer productivity. Previous approaches to dynamic change impact analysis [35, 40, 41] are primarily concerned with the problem of *determining a subset of the methods in a program that were affected by a given set of changes*. That is, they first do a pairwise comparison of high-level program representations, identifying the changes between two program versions, then find all or part of the constructs of the program that are potentially affected by the code changes. Our technique is concerned with the problem of *isolating a subset of the changes that affect a given test*. We developed a series of tools and algorithms to help programmers reduce the amount of time and effort spent in debugging, by determining a safe approximation of the changes responsible for a given test's failure, ranking the changes to indicate the likelihood they may contribute the test's failure, and allowing programmers to experiment with different edits to locate the exact failure causes.

8.1 *Chianti*–the Prototype of Change Impact Analysis

We implemented a prototype – *Chianti*, to perform change impact analysis for Java programs (J2SE 1.4), extending the model of atomic changes and their inter-dependences originally specified in [51].

We presented the experimental validation of the utility of change impact analysis by determining the percentages of affected tests and affecting changes for 40 versions of Daikon in 2002. Our empirical results show that after a program edit, on average the set of affected tests is a bit more than half of all the possible tests (52%) and for each affected test, the number of affecting changes is very small (3.95% of all atomic changes

in that edit). These findings suggest that our change impact analysis is a promising technique for both program understanding and debugging.

The results of our change impact analysis can also indicate the quality of a given test suite. A *good* test suite, when working together with software engineering tools like *Chianti*, should help locate potential program problems as much as possible with reasonable resources usage. In our case, it means more program coverage, less overlap between tests and also smaller tests. These properties will be reflected in the result of our change impact analysis. They will also directly affect the ability or efforts spent on locating problems introduced in the program edit.

First, the test suite should cover the source code as much as possible, such that changes made in the program always affect some tests. Otherwise the edited part of the source code may not covered by any of the test in the given test suite, which limits the ability to discover problems in this part of the program. For example, in the Daikon case study, there were some intervals with atomic changes but none of the existing tests was affected, which is a sign that more tests should be added to cover the edited part of the code.

Second, we expect the overlap between tests in a test suite to be small. That is, each test in the suite should cover a different part of the code. For example, the changes within one component in a multi-component system, should only affect a subset of the whole suite, instead of all the tests in the suite, which often takes long time to rerun.

Last but not least, each test in the test suite should only cover a small part of the code, that is, the size of individual test should be small. This is related to human efforts spent on fault localization. Once a test is affected by some changes and it fails, our system will give the set of affecting changes as the candidates for future investigation. For a specified affected test, we expect a small subset of all the atomic changes are affecting changes, which means less effort will be spent on identifying problematic changes.

8.2 Dependences between Atomic Changes

We defined and implemented (in *Chianti*) finding the dependences between atomic changes. 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. *Crisp* [12, 45] relies on the automated computation of underlying interdependences between atomic changes by *Chianti* to generate the intermediate program from user-specified atomic changes. Three kinds of dependences are defined between atomic changes to ensure the compilability of the intermediate programs. *Structural dependences* capture the necessary sequences that occur when new Java elements are added or deleted in a program. *Declaration dependences* capture all the necessary Java element declarations that are required to create a valid intermediate version. *Mapping dependences* 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. To our knowledge, this is the first such classification of edits of elements of an object-oriented program.

We present initial experiences 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 automatically building compilable intermediate programs given the user-selected atomic changes, and in finding the failure-inducing changes for failing tests.

8.3 Heuristic Ranking of Edits

We also proposed a heuristic to rank the method changes for a failed test, indicating the likelihood that they have contributed to a test failure. Our heuristic is based on the number of ancestors and descendants of a method in the test call graph of the edited program, as well as the calling relationships between changed methods. We evaluated the effectiveness of the heuristic in 14 version pairs from *Eclipse jdt compiler* project. In the case study, we successfully ranked the failure-inducing changes in top 2 in 67% of the failed delegate tests whose failures are caused by a single method change. When

several method changes combined together cause a test to fail, the study shows that in half of the cases, the rankings our heuristic obtains are only one off the ideal rankings. Potentially, this ranking saves programmer from trying all possible combination of the affecting changes. In addition, we also presented other alternative heuristics, discussed why a machine learning technique doesn't work in our case study, and reported the possible ways to use our heuristic to help debugging in a more generalized setting.

8.4 Future Work

One direction for future work is to do an in-depth evaluation of the cost/precision trade-offs involved in using smaller units of change, to better describe change impact to a user, especially since we currently consider all changes to code within a method (i.e., **CM**) as one monolithic change. For example, we may distinguish between the changes to the method declaration and the changes to the method body. This would give users a better understanding about the interface changes and the implementation changes. Further more, we may develop finer-grained changes inside a method body change, allowing more sophisticated dataflow and control analysis inside the changed method, thus providing users a clearer view of the change impact of the edit they made. With the finer-grained changes model, we may be able to perform change impact analysis for security checking analysis. For example, when a programmer changes a method, we should determine the effect of this change on security-sensitive data.

Another possible direction of the future work is to investigate the completeness of the inter-dependences between atomic changes. Currently, we are integrating *Chianti*, *Crisp* and JUnit test framework to automate the entire process of finding failure-inducing changes sets without the need for programmer intervention. *Chianti* is first run to obtain the atomic changes and dependences, as well as the affecting changes for each affected test. Then for each failed test, *Crisp* automatically applies some subset of the affecting changes of the test to build intermediate programs on which the failed test was rerun. The process is repeated until we locate a minimal set of the atomic changes that cause the test to fail. The automation of the process requires the completeness of the inter-dependences between atomic changes. Currently, because of the limitation

of our framework, users need to manually edit part of the Java source code in some special cases.

We also plan to perform experimental comparison of different heuristics for ranking the affecting changes. We have proposed change classifications to categorize the atomic changes with respect to the tests they affect to indicate the likelihood that each change may contribute to a test failure. We also tried heuristics based on the structure of dependence graphs between changes to rank the affecting changes of a failed test. The heuristic we proposed in chapter 6 is based on the calling context of the changed methods. We plan to develop other change classifiers that, for example, take into account the frequency that a change affects a worsening test. We will do a comprehensive comparison of different approaches for locating the failure-inducing changes, including all the heuristics we have already used and random selection.

We are also interested in the application of change impact analysis for helping program understanding by raising the abstraction level. We plan to work on the dependence graph to find some patterns that combine several atomic changes to do certain refactoring changes. We would like to present such information to users giving them a higher level and clearer view about the edit.

References

[1] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering Methodology*, 3(1):3–28, 1994.

[2] Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396, Charleston, South Carolina, United States, 1993.

[3] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 82–91, Shanghai, China, May 2006.

[4] David Binkley. Semantics guided regression test cost reduction. *IEEE Transaction on Software Engineering*, 23(8):498–516, August 1997.

[5] Shawn A. Bohner and Robert S. Arnold. An introduction to software change impact analysis. In Shawn A. Bohner and Robert S. Arnold, editors, *Software Change Impact Analysis*, pages 1–26. IEEE Computer Society Press, 1996.

[6] Jonathan Buckner, Joseph Buchta, Petrenko Maksym, and Vaclav Rajlich. Jripples: A tool for program comprehension during incremental change. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 149–152, St. Louis, MO, May 2005.

[7] Peter Bunus and Peter Fritzson. Semi-automatic fault localization and behavior verification for physical system simulation models. In *ASE '03: Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 253–258, Montreal, Quebec, Canada, October 2003.

[8] Michael Burke and Linda Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, 1993.

[9] Craig Chambers, Jeffrey Dean, and David Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 221–230, Seattle, Washington, United States, 1995.

[10] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: a system for selective regression testing. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 211–220, Sorrento, Italy, 1994.

[11] Ophalia C. Chesley. Crisp - a fault localization tool for java programs. Master's thesis, Rutgers University, October 2007.

[12] Ophelia Chesley, Xiaoxia Ren, and Barbara G. Ryder. Crisp: A debugging tool for Java programs. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 401–410, Budapest, Hungary, September 2005.

[13] Ophelia Chesley, Xiaoxia Ren, Barbara G. Ryder, and Frank Tip. Crisp - a fault localization tool for java programs. In *ICSE '07: Proceeding of the 29th international conference on Software engineering (demo session)*, Minneapolis, MN, USA, May 2007.

[14] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 210–220, Roma, Italy, 2002.

[15] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE '05: Proceeding of the 27th international conference on Software engineering*, pages 342–351, St. Louis, Missouri, USA, May 2005.

[16] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 121–134, San Diego, California, United States, 1996.

[17] Mikhail Dmitriev. Language-specific make technology for the java programming language. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 373–385, Seattle, Washington, USA, 2002.

[18] The Eclipse IDE. http://www.eclipse.org/.

[19] Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine Sorensen, and Mads Tofte. Annodomini: from type theory to year 2000 conversion tool. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, San Antonio, Texas, United States, January 1999.

[20] Sebastian Elbaum, Praveen Kallakuri, Alexey G. Malishevsky, Gregg Rothermel, and Satya Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 12(2), 2003.

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

[22] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transaction on Software Engineering*, 27(2):1–25, February 2001.

[23] Stuart I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–65, 1979.

[24] John R. Foster. *Cost Factors in Software Maintenance*. PhD thesis, University of Durham, Durham, UK., 1993.

[25] Keith Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transaction on Software Engineering*, 17, 1991.

[26] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (Second Edition)*. Addison-Wesley, 2000.

[27] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272, Long Beach, CA, USA, 2005.

[28] Jens Gustavsson. A classification of unanticipated runtime software changes in java. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, pages 4–12, Amsterdam, The Netherlands, 2003.

[29] Steve Gwizdala, Yong Jiang, and Vaclav Rajlich. Jtracker - a tool for change propagation in java. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 223 – 229, Benevento, Italy, 2003.

[30] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 312–326, October 2001.

[31] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, 2002.

[32] Junit, testing resources for extreme programming. http://www.junit.org/.

[33] David Chenho Kung, Jerry Gao, Pei Hsia, F. Wen, Yasufumi Toyoshima, and Cris Chen. Change impact identification in object oriented software maintenance. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 202–211, 1994.

[34] James R. Larus. Whole program paths. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269, Atlanta, Georgia, United States, May 1999.

[35] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, Portland, Oregon, May 2003.

[36] Michelle Lee, A. Jefferson Offutt, and Roger T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, pages 61–70, Santa Barbara, CA, 2000.

[37] James R. Lyle and Mark Weiser. Automatic bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, pages 877–883, Beijing (Peking), China, 1987.

[38] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engg.*, 11(1):7–26, 2004.

[39] Hausi A Muller, Robert Hood, and Ken Kennedy. Efficient recompilation of module interfaces in a software development environment. In *SDE 2: Proceedings of the second ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 180–189, 1987.

[40] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 128–137, Helsinki, Finland, September 2003.

[41] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 491–500, Edinburgh, Scotland, 2004.

[42] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 241–251, Newport Beach, CA, USA, November 2004.

[43] Vaclav Rajlich and Prashant Gosavi. Incremental change in object-oriented programming. *IEEE Software*, 21(4):62–69, 2004.

[44] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132, San Antonio, Texas, United States, January 1999.

[45] Xiaoxia Ren, Ophelia C. Chesley, and Barbara G. Ryder. Identifying failure causes in java programs: An application of change impact analysis. *IEEE Transaction on Software Engineering*, 32(9):718–732, September 2006.

[46] Xiaoxia Ren and Barbara G. Ryder. Heuristic ranking of java program edits for fault localization. In *ISSTA '07: Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis*, London, United Kingdom, July 2007.

[47] Xiaoxia Ren, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 664–665, St. Louis, MO, USA, May 2005.

[48] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 432–448, Vancouver, BC, Canada, October 2004.

[49] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, Ophelia Chesley, and Julian Dolby. Chianti: A prototype change impact analysis tool for Java. Technical Report DCS-TR-533, Rutgers University Department of Computer Science, September 2003.

[50] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transaction on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[51] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, June 2001.

[52] Anita Sarma, Zahra Noroozi, and Andre van der Hoek. Palantir: Raising awareness among configuration management workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 444–454, Portland, Oregon, May 2003.

[53] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: managing the evolution of reusable assets. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 268–285, San Jose, California, United States, 1996.

[54] Maximilian Stoerzer, Barbara G. Ryder, Xiaoxia Ren, and Frank Tip. Finding failure-inducing changes in java programs using change classification. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 57–68, Portland, Oregon, USA, November 2006.

[55] G. Lorenzo Thione. Detecting semantic conflicts in parallel changes, December 2002. Masters Thesis, Department of Electrical and Computer Engineering, University of Texas, Austin.

[56] G. Lorenzo Thione and Dewayne E. Perry. Parallel changes: Detecting semantic interference. Technical report, COMPSAC '05. the 29th Annual International Computer Software and Applications Conference, July 2005.

[57] Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, 1986.

[58] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[59] Paolo Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transaction on Software Engineering*, 29(6):495–509, 2003.

[60] Zhenchang Xing and Eleni Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transaction on Software Engineering*, 31(10):850–868, 2005.

[61] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, Long Beach, CA, USA, November 2005.

[62] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on umldiff change-facts queries. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 263–274, Benevento, Italy, October 2006.

[63] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Philadelphia, Pennsylvania, USA, September 2006.

[64] Zhenchang Xing and Eleni Stroulia. Understanding the evolution and co-evolution of classes in object-oriented systems. *International Journal of Software Engineering and Knowledge Engineering*, 16(1):23–52, 2006.

[65] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, 1999.

[66] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, Charleston, South Carolina, USA, 2002.

[67] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transaction on Software Engineering*, 28(2):183–200, 2002.

Vita

Xiaoxia Ren

2007 Ph. D. in Computer Science, Rutgers University

2000 M.S. in Computer Science, Beijing University, China

1997 B.S. in Computer Science, Beijing University, China

2002-2007 Research assistant, Department of Computer Science, Rutgers University

2001-2002 Teaching assistant, Department of Computer Science, Rutgers University

2000-2001 IT specialist, IBM Global Service, IBM (China) Company Limited.

1997-2000 Research assistant, Department of Computer Science, Beijing University, China

1998-1999 Teaching Assistant, Department of Computer Science, Beijing Univeristy, China

2007 Xiaoxia Ren and Barbara G. Ryder. Heuristic Ranking of Java Program Edits for Fault Localization. In *Proceeding of the International Symposium on Software Testing and Analysis*, July 2007

2007 Ophelia C. Chesley, Xiaoxia Ren, Barbara G. Ryder, Frank Tip. Crisp - A Fault Localization Tool for Java Programs. In *Proceedings of the 29th International Conference on Software Engineering (Formal research demonstrations session)*, MN, May 2007.

2006 Xiaoxia Ren, Ophelia C. Chesley, Barbara G. Ryder. Identifying Failure Causes in Java Programs: an Application of Change Impact Analysis, In *IEEE Transactions on Software Engineering, 32(9), 2006.*

2006 Maximilian Stoerzer, Barbara G. Ryder, Xiaoxia Ren, Frank Tip. Finding Failure- Inducing Changes in Java Programs using Change Classification. In *Proceedings of the 14th SIGSOFT Conference on the Foundations of Software Engineering*, November 2006. (Nominated for best paper award)

2005 Ophelia Chesley, Xiaoxia Ren, Barbara Ryder. Crisp: A Debugging Tool for Java Programs. In *Proceedings of the 21st International Conference on Software Maintenance*, September 2005.

2005 Xiaoxia Ren, Barbara Ryder, Maximilian Stoerzer, Frank Tip. Chianti: A Change Impact Analysis Tool for Java Programs. In *Proceedings of the 27th International Conference on Software Engineering (Formal research demonstrations session)*, May 2005.

2004 Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, Ophelia Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* October 2004.