Chianti: A Tool for Change Impact Analysis of Java Programs

Xiaoxia Ren[1], Fenil Shah[2], Frank Tip[3], Barbara G. Ryder[1], and Ophelia Chesley[1*]

Division of Computer and Information Sciences[1]
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854-8019, USA
{xren,ryder,ochesley}@cs.rutgers.edu

IBM Software Group[2]
17 Skyline Drive
Hawthorne, NY 10532, USA
fenils@us.ibm.com

IBM T.J. Watson Research Center[3]
P.O. Box 704
Yorktown Heights, NY 10598, USA
tip@watson.ibm.com

ABSTRACT

This paper reports on the design and implementation of *Chianti*, a change impact analysis tool for Java that is implemented in the context of the *Eclipse* environment. *Chianti* 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 (regression or unit) tests whose execution behavior may have been modified by the applied changes. For each affected test, *Chianti* also determines a set of *affecting changes* that were responsible for the test's modified behavior. This latter step of isolating the changes that induce the failure of one specific test from those changes that only affect other tests can be used as a debugging technique in situations where a test fails unexpectedly after a long editing session.

We evaluated *Chianti* on a year (2002) of CVS data from M. Ernst's Daikon system, and found that, on average, 52% of Daikon's unit tests are affected. Furthermore, each affected unit test, on average, is affected by only 3.95% of the atomic changes. These findings suggest that our change impact analysis is a promising technique for assisting developers with program understanding and debugging.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.6 [**Software Engineering**]: Programming Environments; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, Measurement, Languages, Reliability

*This research was supported by NSF grant CCR-0204410 and in part by REU supplement CCR-0331797.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

Keywords

Change impact analysis, regression test, unit test, analysis of object-oriented programs

1. INTRODUCTION

The extensive use of subtyping and dynamic dispatch in object-oriented programming languages make it difficult to understand value flow through a program. For example, adding the creation of an object may affect the behavior of virtual method calls that are not lexically near the allocation site. Also, adding a new method definition that overrides an existing method can have a similar non-local effect. This *nonlocality of change impact* is qualitatively different and more important for object-oriented programs than for imperative ones (e.g., in C programs a precise call graph can be derived from syntactic information alone, except for the typically few calls through function pointers [14]).

Change impact analysis [3, 13, 15, 21, 16] consists of a collection of techniques for determining the effects of source code modifications, and can improve programmer productivity by: (i) 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, (ii) reducing the amount of time and effort needed in running regression[1] tests, by determining that some tests are guaranteed not to be affected by a given set of changes, and (iii) 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 [21, 19].

The change impact analysis method presented in this paper presumes the existence of a suite $\mathcal{T}$ of regression tests 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 *interdependent* 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. Then, 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

[1]In the rest of this paper, we will use the term "regression test" to refer to unit tests and other regression tests.

tests, or static call graphs that have been constructed by a static analysis engine. In this paper, we use dynamic call graphs[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 $\mathcal{A}$ against the call graphs for the tests in $\mathcal{T}$ in the *original* version of the program.

4. Finally, 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 $\mathcal{A}$.

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.

There are some interesting similarities and differences between the work presented in this paper, and previous work on regression test selection and change impact analysis. Step 3 above is similar in spirit to previous work on regression test selection. However, unlike previous approaches our technique does not rely on a pairwise comparison of high-level program representations such as control flow graphs (see, e.g. [20]) or Java InterClass Graphs [10]. Our work differs from previous approaches for dynamic change impact analysis [13, 15, 16] in the sense that these previous approaches 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*. In contrast, step 4 of our technique is concerned with the problem of *isolating a subset of the changes that affect a given test*. In addition, our approach decomposes the code edit into a set of semantically meaningful, interdependent 'atomic changes' which can be used to generate intermediate program versions, in order to investigate the cause of unexpected test behavior. These, and other connections to related work, will be further explored in Section 6.

This paper reports on the engineering of *Chianti*, a prototype change impact analysis tool, and 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. [7]. 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. Since the primary goal of our research has been to assist programmers during development, *Chianti* has been integrated closely with Eclipse, a widely used open-source development environment for Java (see `www.eclipse.org`).

The main contributions of this research are as follows:

- Demonstration of the utility of the basic change impact analysis framework of [21], by implementing a proof-of-concept prototype, *Chianti*, and applying it to Daikon, a moderate-sized Java system built by others.

- Extension of the originally specified techniques [21] to handle the entire Java language, including such constructs as anonymous classes and overloading. This work entailed extension of the model of atomic changes and their interdependences.

- 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. 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 Section 2, we give intuition about our approach through an example. In Section 3, the model of atomic changes is discussed, as well as engineering issues arising from handling Java constructs that were previously not modeled. The implementation of *Chianti* is described in Section 4. Section 5 describes the experimental setup and presents the empirical findings of the Daikon study. Related work and conclusions are summarized in Sections 6 and 7, respectively.

2. OVERVIEW OF APPROACH

This section gives an informal overview of the change impact analysis methodology originally presented in [21]. 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. Our method is *safe* [20] in the sense that this set of affected tests contains at least every test whose behavior may have been affected.

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 1(a) to illustrate our approach. Figure 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()`.

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**)[3].

We also compute *syntactic dependences* between atomic changes. Intuitively, an atomic change A_1 is dependent on

[2]Our previous study used static call graphs [19].

[3]There are a few more categories of atomic changes that are not relevant for the example under consideration that will be presented in Section 3.

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. It is important to understand that the syntactic dependences do not capture semantic dependences between changes (consider, e.g., related changes to a variable definition and a variable use in two different methods). This means that if two atomic changes, C_1 and C_2, affect a given test T, then the *absence* of a syntactic dependence between C_1 and C_2 does not imply the absence of a semantic dependence; that is, program behaviors resulting from applying C_1 alone, C_2 alone, or C_1 and C_2 together, may all be different. 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 localize bugs more quickly.

Figure 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 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_2 is dependent on A_1. 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 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 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[4]. 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.

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 paper 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 paper, dynamic call graphs are used.

Figure 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 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.

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 1(d)[5]. 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), (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 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 Fig-

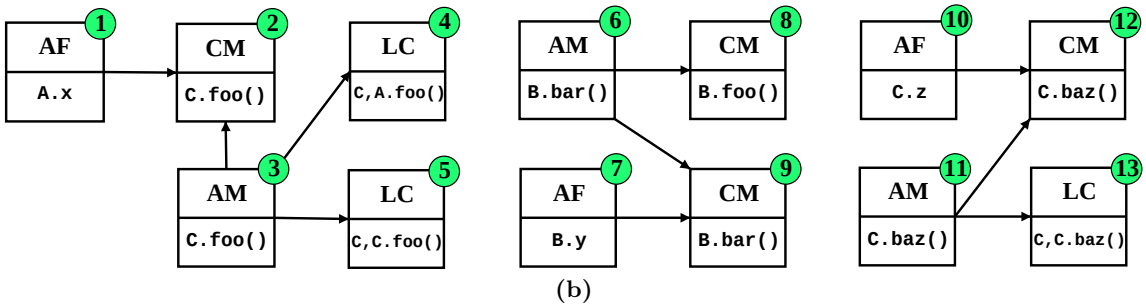
[4]Other scenarios that give rise to **LC** changes will be discussed in Section 3.

[5]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.

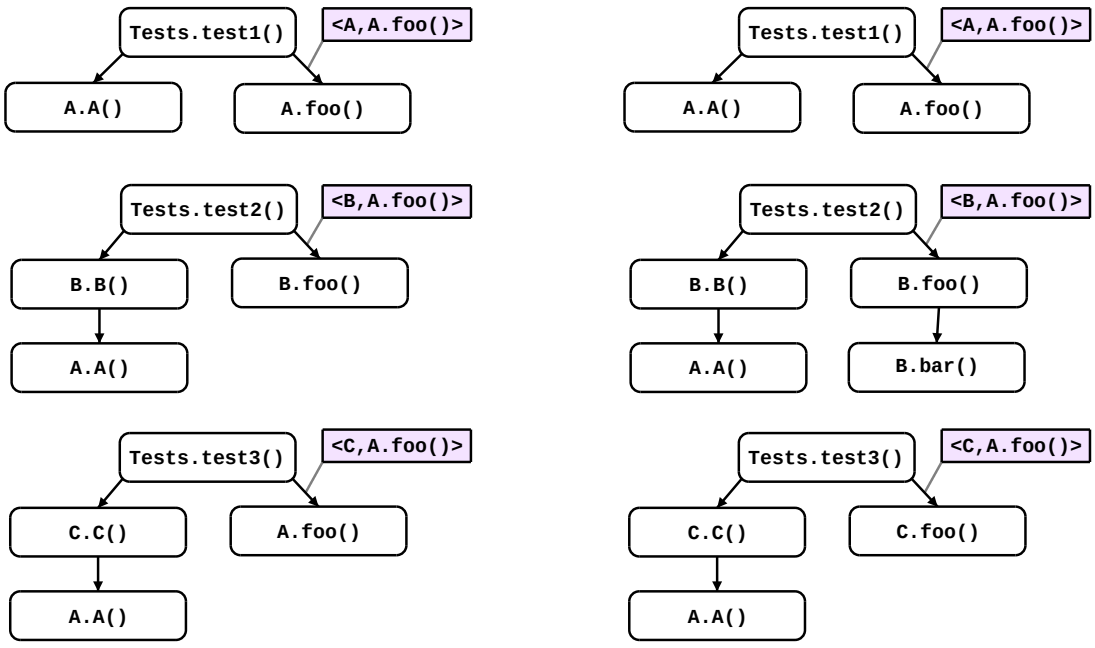
```
class A {
  public A(){ }
  public void foo(){ }
  public int x;
}
class B extends A {
  public B(){ }
  public void foo(){ B.bar(); }
  public static void bar(){ y = 17; }
  public static int y;
}
class C extends A {
  public C(){ }
  public void foo(){ x = 18; }
  public void baz(){ z = 19; }
  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 1: (a) Example program with 3 tests. Added code fragments are shown in boxes. (b) Atomic changes for the example program, with their interdependences. (c) Call graphs for the tests before the changes were applied. (d) Call graphs for the tests after the changes were applied.

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

Figure 2: Affected Tests and Affecting Changes.

ure 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.

We will use the equations in Figure 2 [21] 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 $\mathit{Nodes}(P, t_i)$ and a subset $\mathit{Edges}(P, t_i)$ of calling relationships between P's methods. Likewise, $\mathit{Nodes}(P', t_i)$ and $\mathit{Edges}(P', t_i)$ 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 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. We implicitly make the usual assumptions [10] that program execution is deterministic and that the library code used and the execution environment (e.g., JVM) itself remain unchanged.

3. ATOMIC CHANGES AND THEIR DEPENDENCES

As previously mentioned, a key aspect of our analysis is the step of uniquely decomposing a source code edit into a set of interdependent atomic changes. In the original formulation [21], several kinds of changes, (e.g., changes to access rights of classes, methods, and fields and addition/deletion of comments) were not modeled. Section 3.1 discusses how these changes are handled in *Chianti*. Table 1 lists the set of atomic changes in *Chianti*, which includes the original 8 categories [21] plus 8 new atomic changes (the bottom 8 rows of the table). Most of the atomic changes are self-explanatory except for **CM** and **LC**. **CM** represents any change to a method's body. Some extensions to the original definition of **CM** are discussed in detail in Section 3.1. **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.

3.1 New and Modified Atomic Changes

Chianti handles the full Java programming language, which necessitated the modeling of several constructs not considered in the original framework [21]. Some of these constructs required the definition of new sorts of atomic changes; others were handled by augmenting the interpretation of atomic changes already defined.

AC	Add an empty class
DC	Delete an empty class
AM	Add an empty method
DM	Delete an empty method
CM	Change body of a method
LC	Change virtual method lookup
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

Table 1: Categories of atomic changes.

Initializers, Constructors, and Fields. Six of the newly added changes in Table 1 correspond to initializers. **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. The other two new atomic changes, **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 *Chianti* will report a **CM** for each such constructor, (ii) otherwise, *Chianti* 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**).

Overloading. Overloading poses interesting issues for change impact analysis. Consider the introduction of an overloaded method as shown in Figure 3 (the added method is shown in a box). Note that there are no textual edits in `Test.main()`, 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 `Test.main()` now resolves to `R.foo(Y)` instead of `R.foo(X)` [9]. Therefore, *Chianti* must report a **CM** change for method `Test.main()` despite the fact that no textual changes occur within this method[6].

Hierarchy changes. It is also possible for changes to the

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

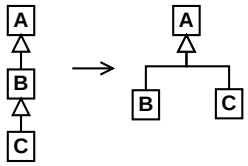
```
class R {
  static void foo(X x){ }
  static void foo(Y y){ }
}
class X { }
class Y extends X { }
class Test {
  static void main(String[] args){
    Y y = new Y();
    R.foo(y);
  }
}
```

Figure 3: Addition of an overloaded method. The added method is shown in a box.

hierarchy to affect the behavior of a method, although the code in the method is not changed. 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 example, in Figure 4(a), method `foo()` contains a cast to type `B`. This cast will succeed if the type of the object pointed to by `a` when execution reaches this statement is `B` or `C` in the original program. In contrast, if we make the hierarchy change shown in Figure 4(b), then this cast will fail if the run-time type of the object which reaches this statement is `C`. Note that the code in method `foo()` has *not changed* due to the edit, but the behavior of `foo()` has been possibly altered. To capture these sorts of changes in behavior due to changes in the hierarchy, we report a **CM** change for the method containing the construct that checks the run-time type of the object (i.e., **CM**($Test.foo()$)).

```
class Test {
  public void foo( ){
    A a = new C();
    ...(B)a...
  }
}
```

(a)



(b)

Figure 4: Hierarchy change that affects a method whose code has not changed.

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 paper 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 paper, this information can be captured by tracing the execution of the tests. If flow-insensitive static analysis is used for constructing call graphs [19], the only significant issue related to threads is to model the implicit calling relationship between `Thread.start()` and `Thread.run()`.

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 also captured as **CM** changes. Possible interprocedural 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.

Changes to CM and LC. 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**. **CM** now 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).

In addition, in some cases, changing a method's access modifier results 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**. Additions and deletions of import statements may also affect dynamic dispatch and are handled by *Chianti*.

3.2 Dependences

Atomic changes have interdependences which induce a partial ordering $\prec$ on a set of them, with transitive closure $\preceq^*$. Specifically, $C_1 \preceq^* C_2$ 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 $D.n()$, there will be a dependence $\mathbf{AM}(C.m()) \prec \mathbf{CM}(D.n())$. Figure 1(b) shows some examples of dependences among atomic changes.

Dependences involving **LC** changes can be caused by edits that alter inheritance relations. **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). For example, making an `abstract` class C non-abstract will result in **LC** changes. In the original dynamic dispatch map, there is no entry with C as the run-time receiver type, but the new dispatch map will contain such an entry. Similar dependences result when other access modifiers are changed.

3.3 Engineering Issues

One engineering problem encountered in building *Chianti* resulted from the absence of unique names for anonymous and local classes. In a JVM, anonymous classes are represented as $EnclosingClassName\$\langle num \rangle$, where the number assigned represents the lexical order of the anonymous class within its enclosing class. This naming strategy guarantees that all the class names in a Java program are unique. However, when *Chianti* compares and analyzes two related Java programs, it needs 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.

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

Figure 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 Java files in a directory that is specified by its parameter. Anonymous 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, *Chianti* uses a naming strategy for classes that assigns each 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*. For the example in Figure 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[7].

4. 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 combination of Eclipse *views*, a *plugin*, and a *launch configuration* that together constitute a change impact analysis *perspective*[8]. *Chianti* is built as a plugin of Eclipse and conceptually can be divided into three functional parts. One part 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[9] in the two project versions. Another part reads test call graphs for the original and edited

[7]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, *Chianti* generates a safe set of atomic changes corresponding to the edit.

[8]A perspective is Eclipse terminology for a collection of views that support a specific task, (e.g., the Java perspective is used for creating Java applications).

[9]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 does not currently include anonymous and local classes.

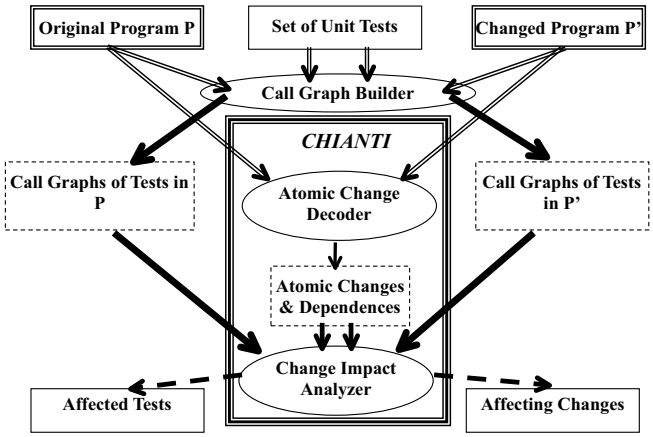


Figure 6: *Chianti* **architecture.**

projects, computes affected tests and their affecting changes. The third part manages the views that allow the user to visualize change impact information. *Chianti*'s GUI also includes a launch configuration that 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. Figure 6 depicts *Chianti*'s architecture.

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[10].

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. When static call graphs are desired, *Chianti* invokes the *Gnosis* analysis engine[11] to construct these [19]. In this case, users need to supply some additional information relevant to the analysis engine (e.g., the choice of call graph construction algorithm to be used and some policy settings for dealing with reflection).

Users can also 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. The experiments with dynamic call graphs presented in this paper have been conducted using an offline tool that instruments the class files for an application. Executing an application that has been instrumented by this tool produces an XML file containing the application's dynamic call graph[12].

[10]See `www.junit.org`.

[11] *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.

[12]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.

When the analysis results are available, the Eclipse workbench changes to the change impact analysis perspective, which provides a number of views:

- 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 7 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 grouped by category.

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).

5. EVALUATION

The experiments with *Chianti* were performed on versions of the Daikon system by M. Ernst et al. [7], 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 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 8 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.

5.1 Atomic Changes

Figure 9(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 5.3 gives more details about two specific intervals in our study.

Figure 9(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, changeInitial-*

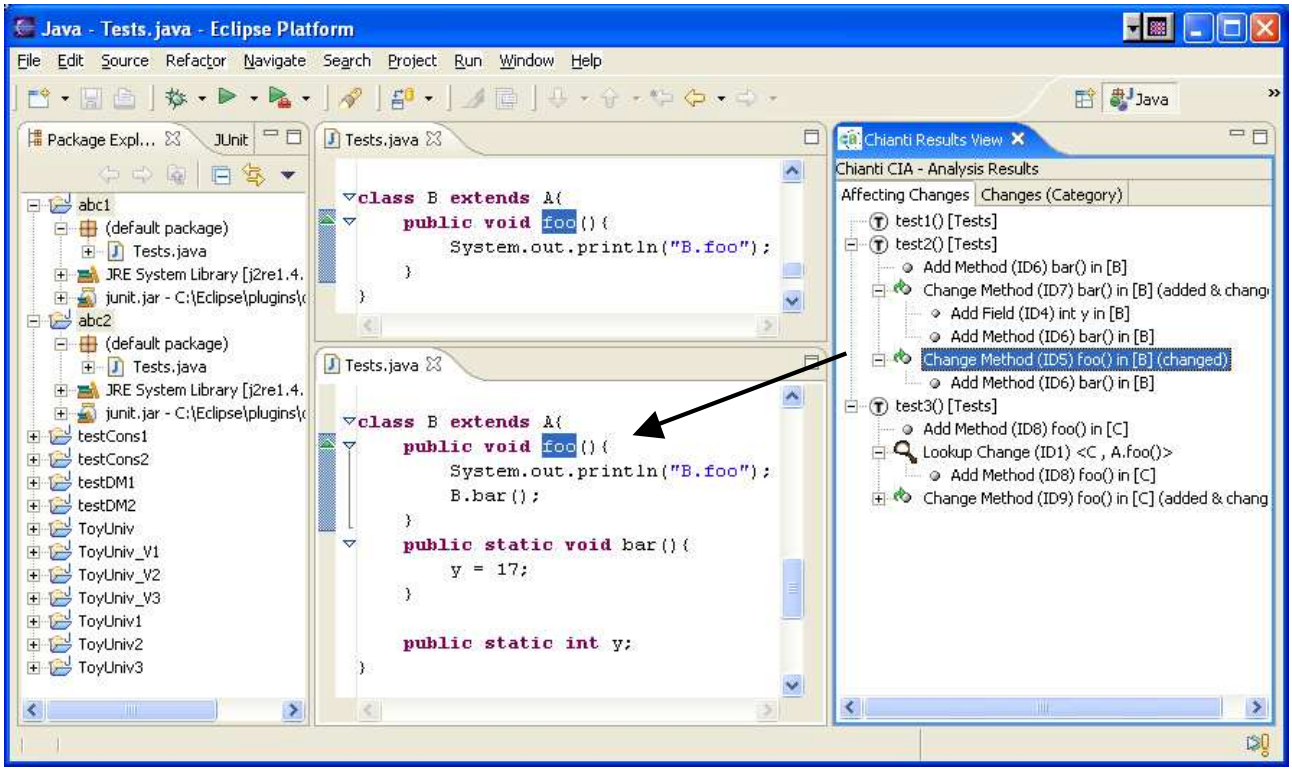


Figure 7: Snapshot of Chianti's *affecting changes* view. The arrow shows how clicking on an atomic change opens an editor on the associated source fragment.

izer and *deleteInitializer*[13]. 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 10 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.

5.2 Affected Tests and Affecting Changes

Figure 11 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 12 shows the average percentage of affecting changes per affected test, for 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, an earlier 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 [19][14]. The closeness of these results to those reported in the present paper suggests that we should investigate the tradeoffs 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 exercised by its unit test suite. For each version of Daikon, we obtained the number of meth-

[13]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.

[14]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.

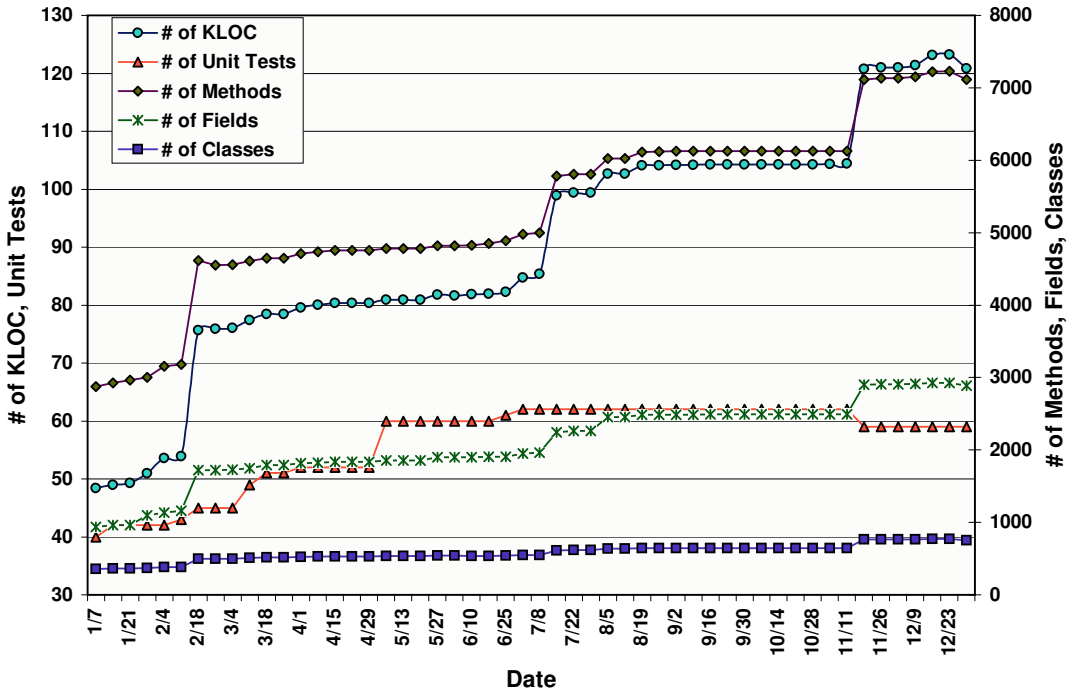


Figure 8: Daikon growth statistics for the year 2002

ods 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.

5.3 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.

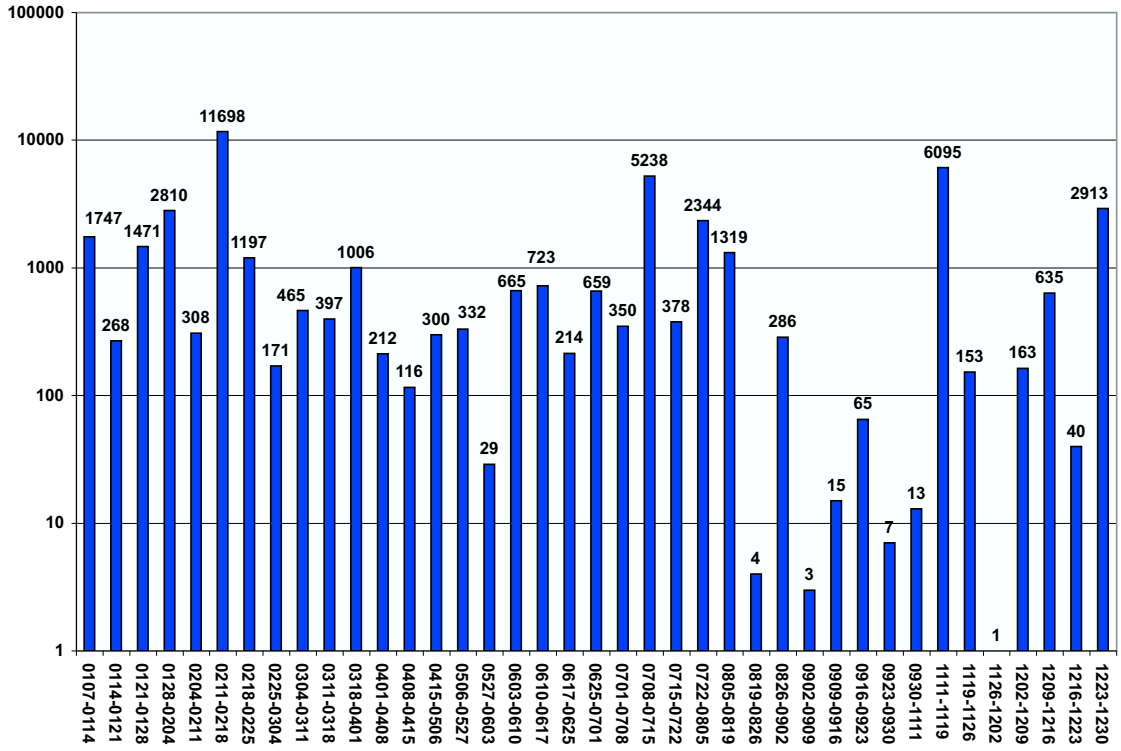
Case Study 1 The first interval we decided to explore further is the one for which we found the highest percentage of affected tests. This occurred between versions 07/08/02 and 07/15/02, when 88.7% (55 out of 62) of the tests were affected. We partitioned the version history of this interval into daily 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 13(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).

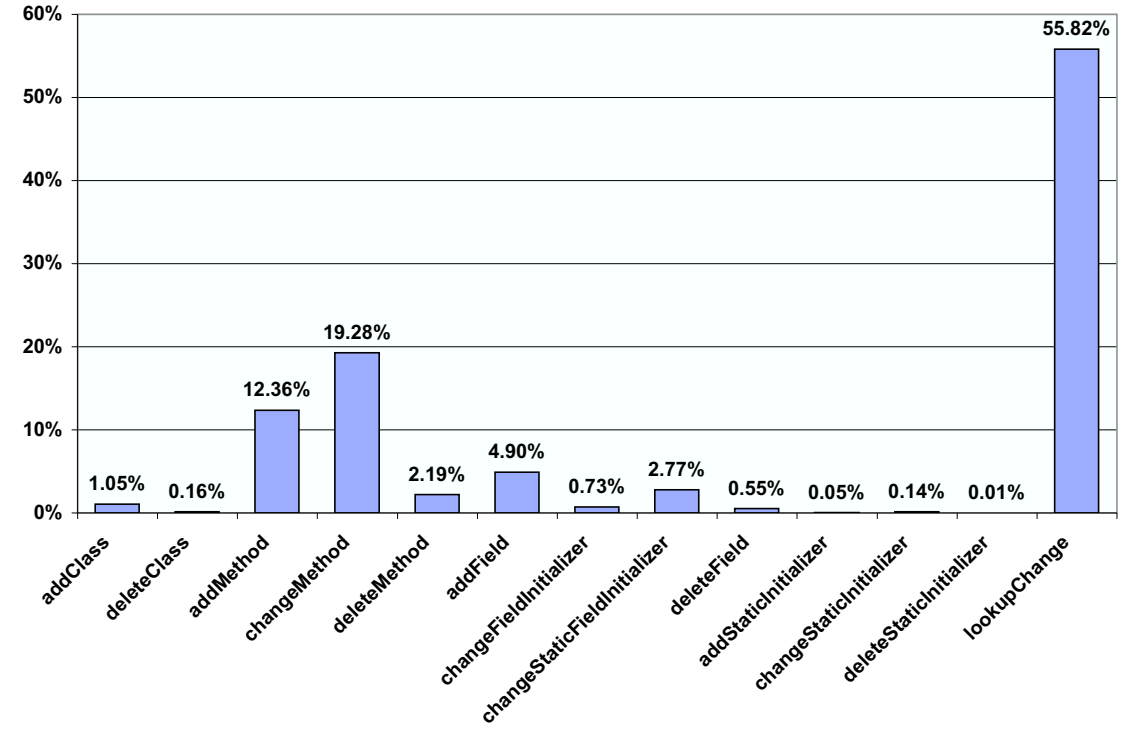
Figure 13(b) shows the total number of atomic changes and the average number of affecting changes per affected test in each subinterval compared with the original interval (again shown as the rightmost pair of bars). The use of smaller intervals resulted in smaller numbers of atomic changes for each interval and also smaller numbers of affecting changes per affected test; this makes the tracing of 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 14(a) and (b) we can see similar results to those of case study 1, that is, we obtain 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 improves 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



(a)



(b)

Figure 9: (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.

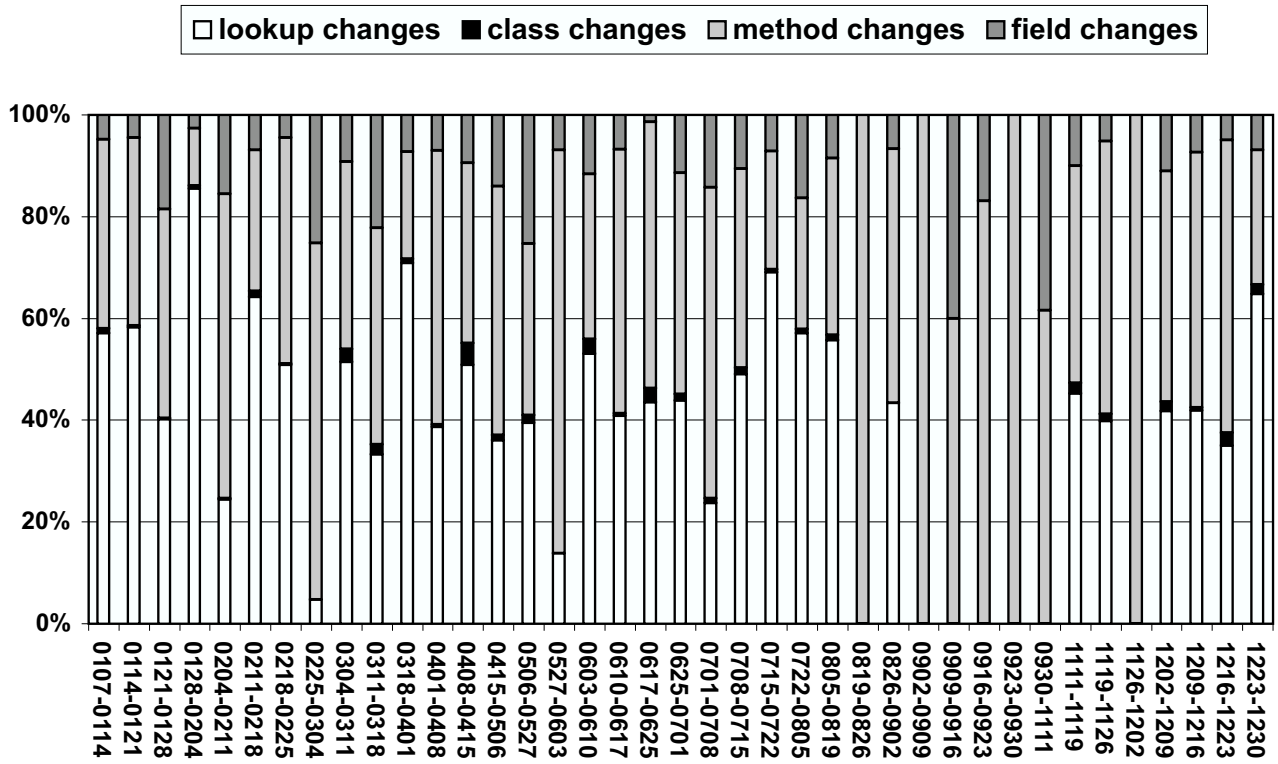


Figure 10: Classification of atomic changes for each pair of versions. Class changes include AC and DC. Method changes include AM, CM, DM, ASI, DSI and CSI. Field changes include AF, DF, CSFI and CFI.

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 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.

5.4 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. 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.

6. RELATED WORK

In previous papers, we presented the conceptual framework of our change impact analysis, without empirical experimentation [21], and reported on experiments with a purely static version of *Chianti* using static call graphs generated by *Gnosis*, on the same Daikon data used here [19].

We distinguish three broad categories of related work in the community: (i) change impact analysis techniques, (ii) regression test selection techniques, and (iii) techniques for controlling the way changes are made.

6.1 Change Impact Analysis Techniques

Previous research in change impact analysis has varied from approaches relying completely on static information, including the early analyses of [3, 11], to approaches that only utilize dynamic information, such as [13]. There also are some methods [15] 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 of the impact analyses previous to ours focus 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

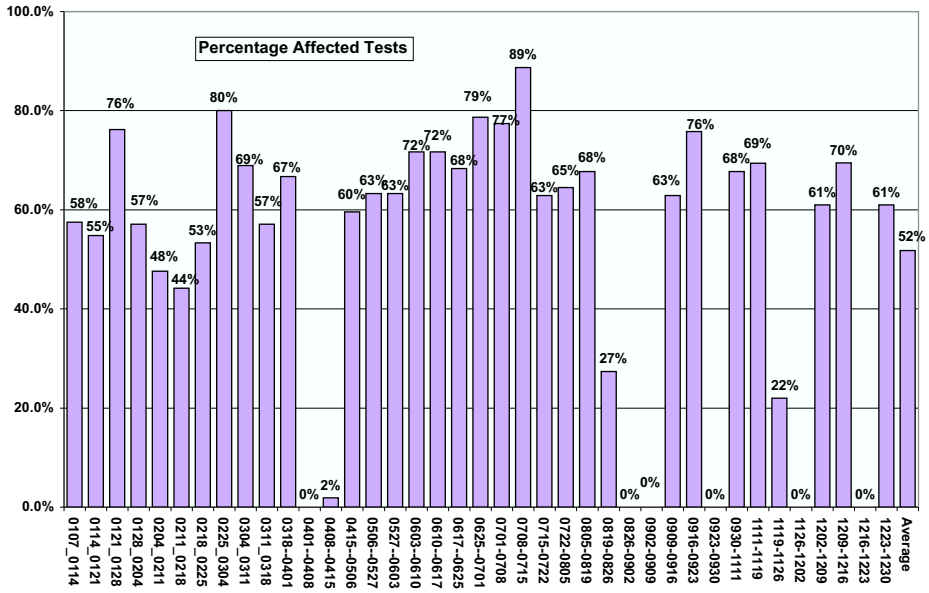


Figure 11: Percentage of affected tests for each of the Daikon versions.

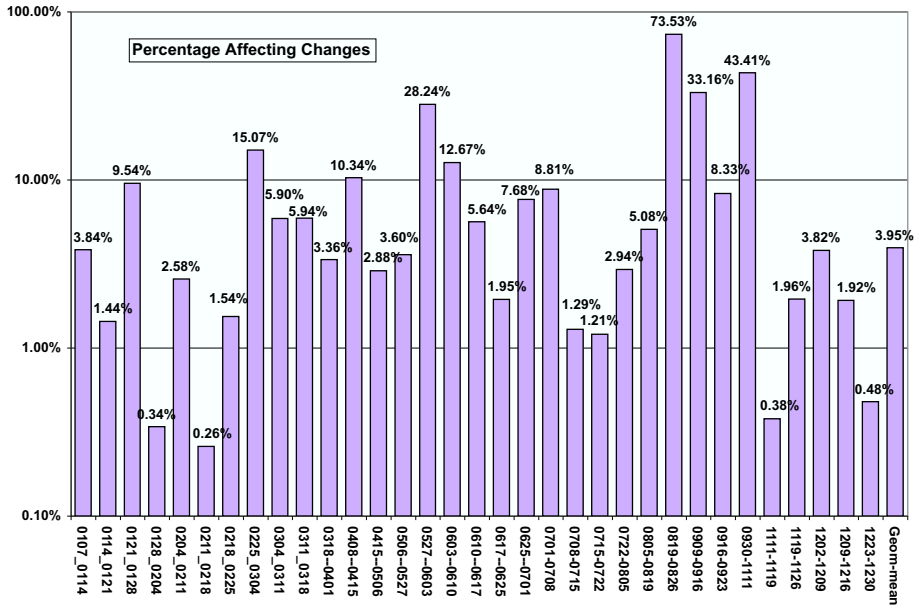
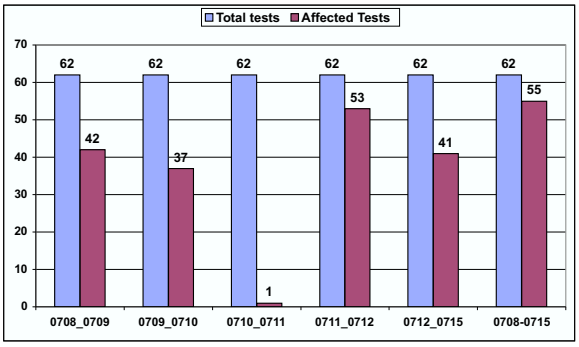
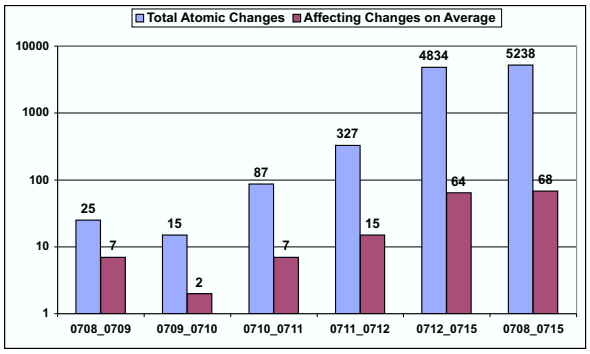


Figure 12: Average percentage of affecting changes, per affected test, for each of the Daikon versions.

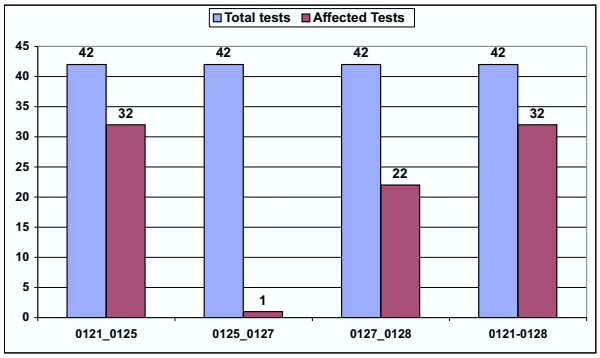


(a) Number of affected tests on large and smaller daily intervals.

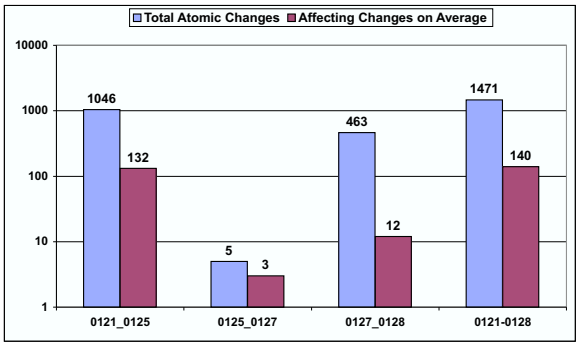


(b)Number of atomic changes and on average affecting changes on large interval and smaller daily intervals (note log scale)

Figure 13: Detailed analysis results for the interval 7/08/02—7/15/02.



(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 14: Detailed analysis results for the interval 1/21/02—1/28/02.

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[15] was presented by Bohner and Arnold [3] as "intuitively appealing" and "a starting point" for implementing change impact analysis tools. However, 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.* [11] 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.

Tonella's impact analysis [27] 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 [8]. 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 [5, 18] 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. [25, 24] wish to find possible semantic interferences 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. [15] uses a combined methodology, by correlating a forward static slice [26] with respect to a changed program entity (i.e., a basic block or method) with execution data obtained from instrumented applications. Each program en-

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

tity 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. [15] 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 codes, 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 [13] present *PathImpact*, a dynamic impact analysis that is based on whole-path profiling [12]. 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. 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 [16] of the dynamic impact analyses *CoverageImpact* by Orso et al. [15] and *PathImpact* by Law and Rothermel [13] 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* [16] 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.

Zeller [28] introduced the delta debugging approach for localizing failure-inducing changes among large sets of textual changes. Efficient binary-search-like techniques are used to partition changes into subsets, executing the programs resulting from applying these subsets, and determining whether the result is correct, incorrect, or inconclusive. An important difference with our work is that our atomic changes and interdependences take into account program structure and dependences between changes, whereas Zeller assumes all changes to be completely independent.

6.2 Regression Test Selection

Selective regression testing[16] aims at reducing the number of regression tests that must be executed after a software change [20, 17]. 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* [4] uses a notion of module-level coverage, and *DejaVu* [20] 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 [1] and Binkley [2] 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 to our work, this work uses more costly static analyses 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 [20] present a regression test selection technique that relies on a simultaneous traversal of two program representations (control flow graphs (CFGs) in [20]) to identify those program entities (edges in [20]) 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. [10] present a safe regression test selection technique for Java that is an adaptation of the technique of Rothermel and Harrold [20]. 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.

The method for finding affected tests presented in this paper is also *safe* in the sense that it is guaranteed to identify any test that reveals a fault. However, unlike the regression test selection techniques such as [20, 10], our method 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 tradeoffs 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. [6], 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

[16] 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.

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 [15] 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.

6.3 Controlling the Change Process

Palantir [22] 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 [23] 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.

7. CONCLUSIONS AND FUTURE WORK

We have presented our experiences with *Chianti*, a change impact analysis tool that has been validated on a year of CVS data from Daikon. 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.

Plans for future research include an in-depth evaluation of the cost/precision tradeoffs involved in using static versus dynamic call graphs, now that we have some experience with both. We also intend to experiment with 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.

Acknowledgements. We would like to thank Michael Ernst and his research group at MIT for the use of their data. We are also grateful to the anonymous reviewers for their constructive feedback.

8. REFERENCES

[1] Bates, S., and Horwitz, S. Incremental program testing using program dependence graphs. In *Proc. of the ACM SIGPLAN-SIGACT Conf. on Principles of Programming Languages (POPL'93)* (Charleston, SC, 1993), pp. 384–396.

[2] Binkley, D. Semantics guided regression test cost reduction. *IEEE Trans. on Software Engineering 23*, 8 (August 1997).

[3] Bohner, S. A., and Arnold, R. S. An introduction to software change impact analysis. In *Software Change Impact Analysis*, S. A. Bohner and R. S. Arnold, Eds. IEEE Computer Society Press, 1996, pp. 1–26.

[4] Chen, Y., Rosenblum, D., and Vo, K. Testtube: A system for selective regression testing. In *Proc. of the 16th Int. Conf. on Software Engineering* (1994), pp. 211–220.

[5] Eidorff, P. H., Henglein, F., Mossin, C., Niss, H., Sorensen, M. H., and Tofte, M. AnnoDomini: From type theory to year 2000 conversion. In *Proc. of the ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (January 1999), pp. 11–14.

[6] Elbaum, S., Kallakuri, P., Malishevsky, A. G., Rothermel, G., and Kanduri, S. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification, and Reliability* (2003). To appear.

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

[8] Gallagher, K., and Lyle, J. R. Using program slicing in software maintenance. *IEEE Trans. on Software Engineering 17* (1991).

[9] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification (Second Edition)*. Addison-Wesley, 2000.

[10] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A. Regression test selection for Java software. In *Proc. of the ACM SIGPLAN Conf. on Object Oriented Programming Languages and Systems (OOPSLA'01)* (October 2001), pp. 312–326.

[11] Kung, D. C., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., and Chen, C. Change impact identification in object oriented software maintenance. In *Proc. of the International Conf. on Software Maintenance* (1994), pp. 202–211.

[12] Larus, J. Whole program paths. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation* (May 1999), pp. 1–11.

[13] Law, J., and Rothermel, G. Whole program path-based dynamic impact analysis. In *Proc. of the International Conf. on Software Engineering* (2003), pp. 308–318.

[14] Milanova, A., Rountev, A., and Ryder, B. G. Precise call graphs for C programs with function pointers. *Journal for Automated Software Engineering* (2004). Special issue on *Source Code Analysis and Manipulation*.

[15] Orso, A., Apiwattanapong, T., and Harrold, M. J. Leveraging field data for impact analysis and regression testing. In *Proc. of European Software Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE'03)* (Helsinki, Finland, September 2003).

[16] Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., and Harrold, M. J. An empirical comparison of dynamic impact analysis algorithms. In *Proc. of the International Conf. on Software Engineering (ICSE'04)* (Edinburgh, Scotland, 2004), pp. 491–500.

[17] Orso, A., Shi, N., and Harrold, M. J. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)* (Newport Beach, CA, 2004). To appear.

[18] Ramalingam, G., Field, J., and Tip, F. Aggregate structure identification and its application to program analysis. In *Proc. of the ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (January 1999), pp. 119–132.

[19] Ren, X., Shah, F., Tip, F., Ryder, B. G., Chesley, O., and Dolby, J. Chianti: A prototype change impact analysis tool for Java. Tech. Rep. DCS-TR-533, Rutgers University Department of Computer Science, September 2003.

[20] Rothermel, G., and Harrold, M. J. A safe, efficient regression test selection technique. *ACM Trans. on Software Engineering and Methodology 6*, 2 (April 1997), 173–210.

[21] Ryder, B. G., and Tip, F. Change impact for object oriented programs. In *Proc. of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis and Software Testing (PASTE01)* (June 2001).

[22] Sarma, A., Noroozi, Z., and van der Hoek, A. Palantir: Raising awareness among configuration management workspaces. In *Proc. of the International Conf. on Software Engineering* (2003), pp. 444–454.

[23] Steyaert, P., Lucas, C., Mens, K., and D'Hondt, T. Reuse contracts: Managing the evolution of reusable assets. In *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1996), pp. 268–285.

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

[25] Thione, G. L., and Perry, D. E. Parallel changes: Detecting semantic interference. Tech. Rep. ESEL-2003-DSI-1, Experimental Software Engineering Laboratory, University of Texas, Austin, September 2003.

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

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

[28] Zeller, A. Yesterday my program worked. Today, it does not. Why? In *Proc. of the 7th European Software Engineering Conf./7th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE'99)* (Toulouse, France, 1999), pp. 253–267.