Chianti: A Prototype Change Impact Analysis Tool for Java

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

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
{ftip,dolby}@us.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 reported in terms of affected 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. We evaluated Chianti on 6 months of data from M. Ernst's Daikon system, and found that, on average, 62.4% of the tests is affected. Furthermore, each affected test, on average, is affected by only 5.6% of the atomic changes. These findings suggest that change impact analysis is a promising technique for assisting developers with program understanding and debugging.

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).

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

Change impact analysis [3, 12, 13, 17] consists of a collection of techniques for determining the effects of source code modifications, and can improve programmer productivity in several ways. First, change impact analysis allows programmers to experiment with different edits, observe their effect, and use this information to determine which edit to select. Change impact analysis may also reduce the amount of time and effort needed in running (unit or regression) tests, by determining that some tests are guaranteed not to be affected by a given set of changes. Finally, change impact analysis [17] may 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.

Previous work on change impact analysis is either based on dynamic execution data and thus captures impact with regard to a particular execution only [12, 13], or it uses a reachability measure on static call graphs that is quite imprecise [3]. In contrast, the analysis studied in this paper [17] is based on reasonably precise static call graphs (computed using the 0-CFA algorithm [9, 16]), and reports possible impact over all program executions.

Our analysis comprises the following steps. First, a source code edit is analyzed to obtain a set of interdependent atomic changes $\mathcal{A}$, whose granularity is (roughly speaking) at the method level, so that it matches that of the static analysis we use. Second, for a given set $\mathcal{T}$ of (unit or 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 $\mathcal{T}$. Third, for a given test $t_i \in \mathcal{T}'$, the analysis finds a subset $\mathcal{A}'$ of $\mathcal{A}$ that contains all the changes that affect t_i, by correlating the call graph for t_i with the changes in $\mathcal{A}$.

This paper reports on the engineering of *Chianti*, a prototype change impact analysis tool, and its validation against a 6-month long revision history (taken from the developers' CVS repository) of *Daikon*, a large, realistic Java system developed by M. Ernst et al. [7]. 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 [17], 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 [17] to handle the entire Java language, including such constructs as anonymous classes. This work entailed extension of the model of atomic changes and their interdependences.

- Experimental validation of the usefulness of change impact analysis by determining the percentages of affected tests and affecting changes for 21 versions of Daikon. For the 20 sets of changes between these versions, we found that, on average, 62.4% of the tests are potentially affected. Moreover, for each potentially affected test, on average, only 5.9% of the changes affected it.

Clearly, the small number of affecting changes per affected test indicates that our techniques are useful for program understanding and debugging. The performance of *Chianti* has thus far not been our primary focus, and the running time of the tool currently is disappointing. However, we have several concrete ideas for improving performance (discussed in Section 4) that we are pursuing.

In Section 2, the model of atomic changes is discussed, as well as engineering issues arising from handling Java constructs that were previously not modeled. *Chianti*'s implementation is described in Section 3. Section 4 describes the experimental setup and presents the empirical findings of the Daikon study. Related work and conclusions are summarized in Sections 5 and 6, respectively.

2 Atomic Changes and Their Dependences

Chianti is based on a conceptual framework for change impact analysis of object-oriented programs that was originally presented at PASTE'01 [17]. This framework assumes that an original program P is edited to yield a changed program P', where both P and P' are syntactically correct and compilable. The edit itself is decomposed into a set $\mathcal{A}$ of *atomic changes* such as "add an empty class'' or "delete a method''. Associated with P is a set of tests $\mathcal{T}$ t_{1},...,t_n. Each t_i exercises a subset *Nodes*(P, t_i) of P's methods, and a subset *Edges*(P, t_i) of calling relationships between P's methods, which form a call graph[1] G_{t_i}. 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 $A.m() \rightarrow_C B.n()$, indicating possible control flow from method $A.m()$ to method $B.n()$ due to a virtual call to method $n()$ on an object of type C.

As 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 [17], several kinds of changes, (e.g., changes to access rights of classes, methods, and fields and addition/deletion of comments) were not modeled. Section 2.1 discusses how these changes are handled in *Chianti*. Table 1 lists the set of atomic changes in *Chianti*, which includes the original 8 categories of [17] plus 4 new atomic changes (the bottom 4 rows of the table). Most of the atomic changes are self-explanatory except for **CM** and **LC**. **CM** represents any change to a method. Some extensions to the original definition of **CM** are discussed in detail in Section 2.1. **LC** represents any source code change that affects dynamic dispatch behavior. For example, the addition/deletion of methods, changes to the access control of methods, or addition/deletion of inheritance relations may have this effect. **LC** is defined as a set of pairs $\langle C, A.m() \rangle$, indicating that the dynamic dispatch behavior for a call to $A.m()$ on an object with run-time type C has changed.

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
AI	Add an empty initializer
DI	Delete an empty initializer
CI	Change definition of initializer
CF	Change definition of a field initializer

Table 1. Categories of atomic changes.

Once atomic changes have been computed, the analysis proceeds in two steps. First, it determines the *affected tests* t_i whose call graph G_{t_i} contains nodes that have been changed or deleted, or edges that correspond to changed dispatch relationships. Then, for a given affected test t_i, further analysis can determine a safe approximation of the *affecting changes* that may have influenced t_i's behavior. Figure 1 provides definitions of these notions of *AffectedTests* and

[1] In this paper, static call graphs are used. Dynamic call graphs could be used as well, but this would limit the validity of the results to a (set of) specific executions.

$$
\begin{array}{ll}
\textit{AffectedTests}(\mathcal{T},\mathcal{A}) & \{\, t_{\ i} \mid t_i \in \mathcal{T},\ \textit{Nodes}(P,t_i) \cap (\mathbf{CM} \cup \mathbf{DM}))\ \ /\emptyset \,\} \cup \\
& \{\, t_i \mid t_i \in \mathcal{T},\ n \in \textit{Nodes}(P,t_i),\ n {\rightarrow_B} A.m \in \textit{Edges}(P,t_i),\ \langle B, X.m\rangle \in \mathbf{LC},\ B{<^*}A{\leq^*}X \,\} \\
\\
\textit{AffectingChanges}(t,\mathcal{A}) & \{\, a\ ' \mid a \in \textit{Nodes}(P',t) \cap (\mathbf{CM} \cup \mathbf{AM}),\ a' \preceq^* a \,\} \cup \\
& \{\, a' \mid\ a \equiv \langle B, X.m\rangle \in \mathbf{LC},\ B{<^*}A{\leq^*}X,\ n {\rightarrow_B} A.m \in \textit{Edges}(P',t), \\
& \qquad \text{for some } n, A.m \in \textit{Nodes}(P',t),\ a' \preceq^* a \,\}
\end{array}
$$

Figure 1. Affected Tests and Affecting Changes.

AffectingChanges [17].

2.1 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 [17]. 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.

Initializers, Constructors, and Fields. Three of the newly added changes in Table 1 correspond to initializers. **AI** and **DI** denote the set of added and deleted initializers respectively, which can be *instance* or *static* initializers. **CI** captures any change to an initializer. The fourth new atomic change, **CF**, captures any change to a 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).

However, 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 non-static field initializations or 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 (i.e., **CF**, **CI**) to a *static* field or *static* initializer.

Overloading. Overloading poses interesting issues for change impact analysis. Consider the introduction of an overloaded method as shown in Figure 2. 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)`. Therefore, *Chianti* must report a **CM** change for method `Test.main()` despite the fact that no textual changes occur within this method[2].

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

Figure 2. Addition of an overloaded method.

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 corresponds to: (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 (including modifier changes on the method header).

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. Additions and deletions of import statements and changes to the class hierarchy may also affect dynamic dispatch and are handled by *Chianti*.

2.2. Dependences

Atomic changes have interdependences which induce a partial ordering $\prec$ on a set of them, with transitive closure $\preceq^*$. This determines a safe order in which atomic changes

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

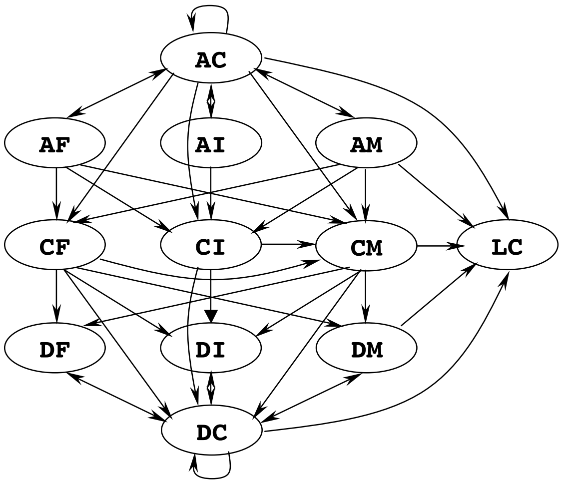

Figure 3. Atomic change dependences.

can be applied to P to obtain an 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 (therefore **AC**$(X) \prec$ **AM**$(X.m())$ and **AC**$(X) \prec$ **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 **AM**$(C.m()) \prec$ **CM**$(D.n())$. The full set of dependence relations possible between atomic changes is shown in Figure 3.

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 (caused by declaring a new class/interface or method), (ii) deleted dynamic dispatch tuples (caused by deleting a class/interface or method), and (iii) dynamic dispatch tuples with changed targets (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 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. Similar dependences result when other access modifiers are changed.

2.3 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 inner class within its enclosing class. This naming strategy guarantees that all the class names in a Java program are unique. However, *Chianti* compares and analyzes two related Java programs, and needs to establish a correspondence between classes and methods in each version. 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 local or anonymous classes, this strategy requires further consideration.

```
import java.io.*;
class Lister
{//code added by edit********
  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]);
  }//end of code added by edit*********
  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 4. Addition of an anonymous class.

Figure 4 shows a simple program using anonymous classes with the code added by the edit indicated by comments. 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 4, 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 local classes and report atomic changes involving them[3].

3. Implementation

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 *plugins*, *views*, and a *launch configuration* that together constitute a change impact analysis *perspective*[4]. One plugin is responsible for deriving a set of atomic changes from two versions of a project, which is achieved via a pairwise comparison of the abstract syntax trees of the classes[5] in the two versions of project. Another plugin communicates with an external static analysis engine that computes call graphs, and computes affected tests and affecting changes. This plugin also manages the various views that visualize change impact information. *Chianti*'s GUI also includes a launch configuration that allows users to select the projects versions to be analyzed, the set of tests associated with the project, and the call graph construction algorithm to be used. Figure 5 depicts *Chianti*'s architecture.

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

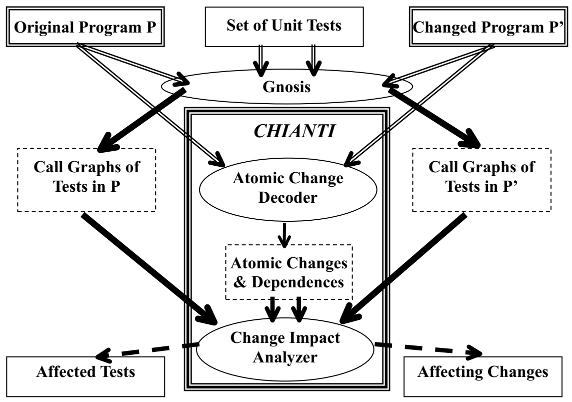

Figure 5. *Chianti* **architecture.**

as the test suite associated with these projects (currently, we allow tests that have a separate `main()` routine and *JUnit* tests[6]). Some information relevant to the static analysis engine must also be supplied (e.g., the call graph construction algorithm to be used, and some policy settings for dealing with reflection). After entering this information, the analysis is started. When the analysis results are available, the Eclipse workbench changes perspective to the change impact analysis perspective, which provides a number of views:

- The *atomic-changes-by-category view* shows the different atomic changes grouped by category. Each atomic change is the root of a tree that can be expanded on demand to show dependent changes. This quickly provides an idea of the different "threads'' of changes that have occurred. Figure 6 shows a snapshot of this view.

- The *atomic-changes-by-type view* shows the different atomic changes grouped by the class that contains the change. This grouping provides an idea of how the changes are distributed over the program.

- The *affected tests view* shows the affected tests in a tree view, with the associated methods and call sites as children of the affected test.

- The *affecting changes view*, which is in the process of being implemented, will consist of a tree view of the affected tests, with the atomic changes that affected it as children in the tree.

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 *atomic-changes-by-category view* opens an editor on the associated program fragment).

[3]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.

[4]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).

[5]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.

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

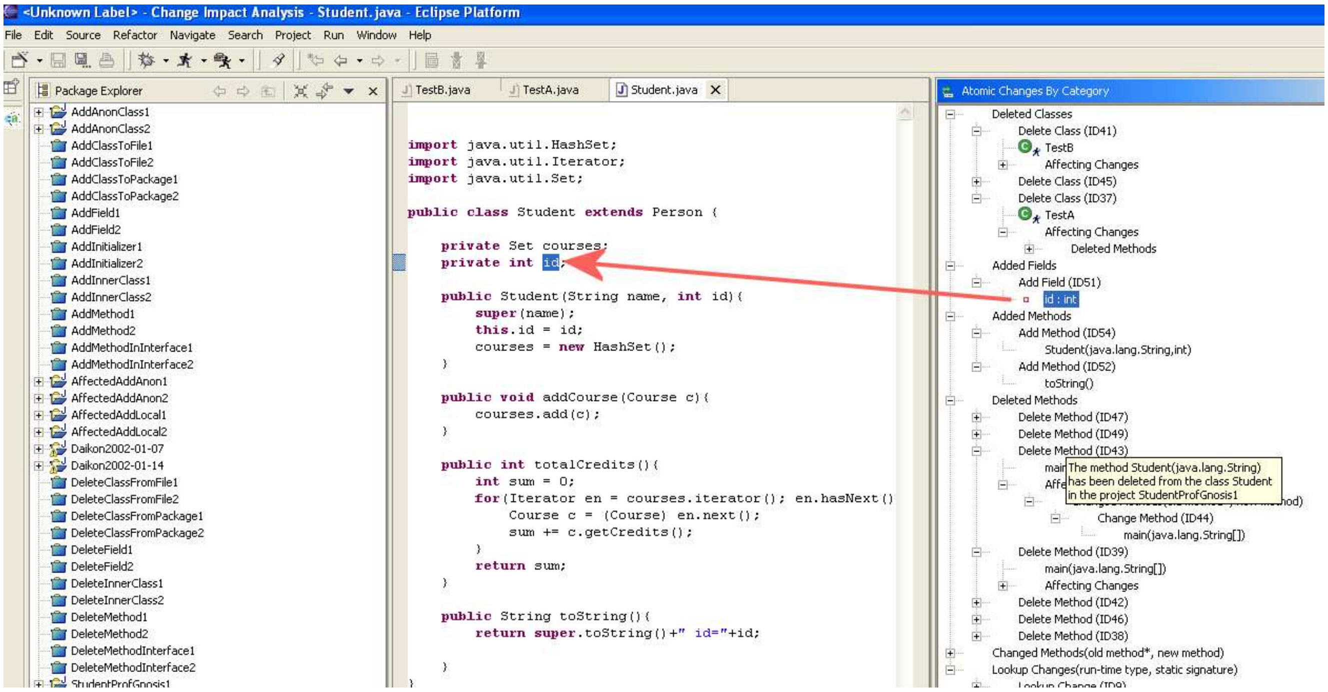

Figure 6. Snapshot of *Chianti*'s *atomic-changes-by-category* view. The arrow shows how clicking on an atomic change opens an editor on the associated source fragment.

We use the *Gnosis* analysis engine to construct the call graphs and type information that are required for our analysis using the 0-CFA algorithm [9, 16]. *Gnosis* is being developed at IBM Research as a test-bed for research on demand-driven and context-sensitive static analysis. One of the strengths of Gnosis is its extensive modeling of the native code in the Java libraries and of the reflective aspects of the Java language. This is essential for analyzing the Daikon system, which relies heavily on reflection. We validated the correctness of the computed call graphs by instrumenting the Daikon code with assertions that compare the types of objects that occur at run-time against the static estimates computed by Gnosis.

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.

4 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 version history arbitrarily at week boundaries. All modifications checked in between one week boundary and the next were considered 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 with 1 week, until it included changes. The data reported in this section covers the first 6 months (i.e., 26 weeks) of updates, during which there were 20 intervals with editing activity.

During the 6-month period under consideration, Daikon was actively being developed, and increased in size from 48K to 82K lines of code. More significant are the program-based measures of growth, from 357 to 542 classes, 2409 to 4339 methods, and 937 to 1906 fields. The number of unit tests associated with Daikon grew from 40 to 62 during the time period under consideration. Figure 7 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 6 month period.

Atomic changes. Figure 8 shows the number of atomic changes between each pair of versions. The number of atomic changes per interval varies greatly between 31 and 11,887 during this period, although only 6 edits involved more than 1,000 atomic changes. Investigation of the largest edit revealed that during this week a parser was added to the system, which involved the addition of 100+ classes.

Figure 9 summarizes the relative percentages of kinds of atomic changes observed during the entire 6-month period. The height of each cone indicates the frequency of the cor-

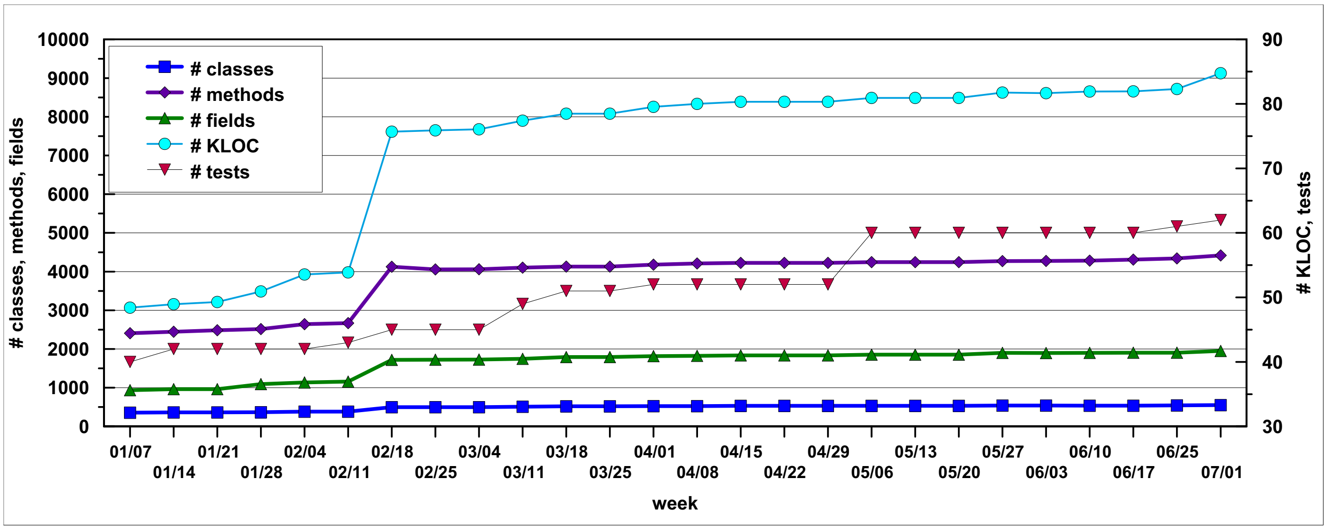

Figure 7. Daikon growth statistics for the 26 weeks between 01/01/02 and 07/01/02.

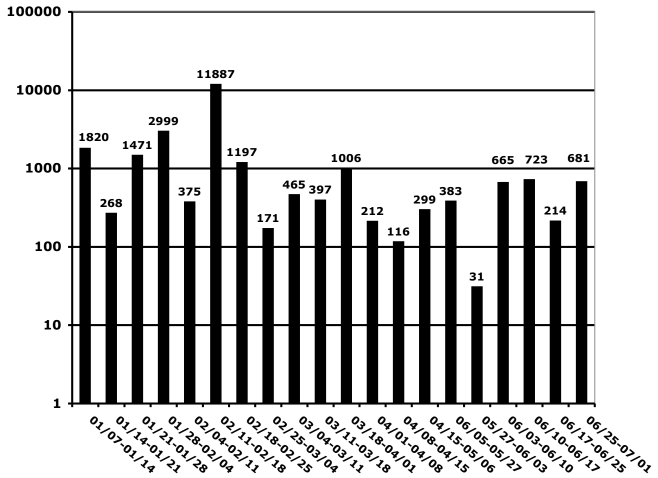

Figure 8. Number of atomic changes between each pair of versions (note log scale).

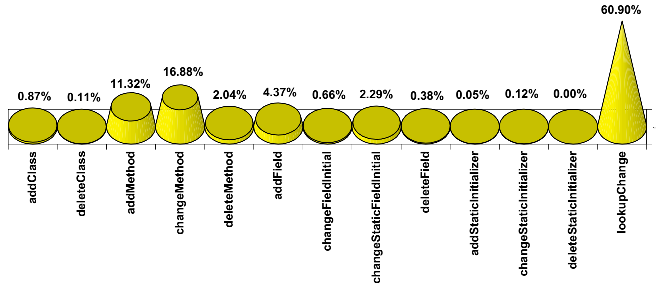

Figure 9. Breakdown of the kinds of atomic changes over the entire 6-month period.

Figure 10. Classification of atomic changes for each pair of versions.

responding kind of atomic change; these values vary widely, by three orders of magnitude. Note that the 0.0% value for *deleteStaticInitializer* in the figure is not actually zero, but represents the 1 atomic change of that type in a total of over 25,000 changes for the entire period!

Figure 10 shows the proportion of atomic changes per interval, grouped by the program construct they affect, namely, classes, fields, methods and dynamic dispatch (i.e., lookup change). 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.

Affected tests. Figure 11 shows the percentage of affected tests for each of the Daikon versions. On average, 62.4% of the tests is affected in each edit. Interestingly, no

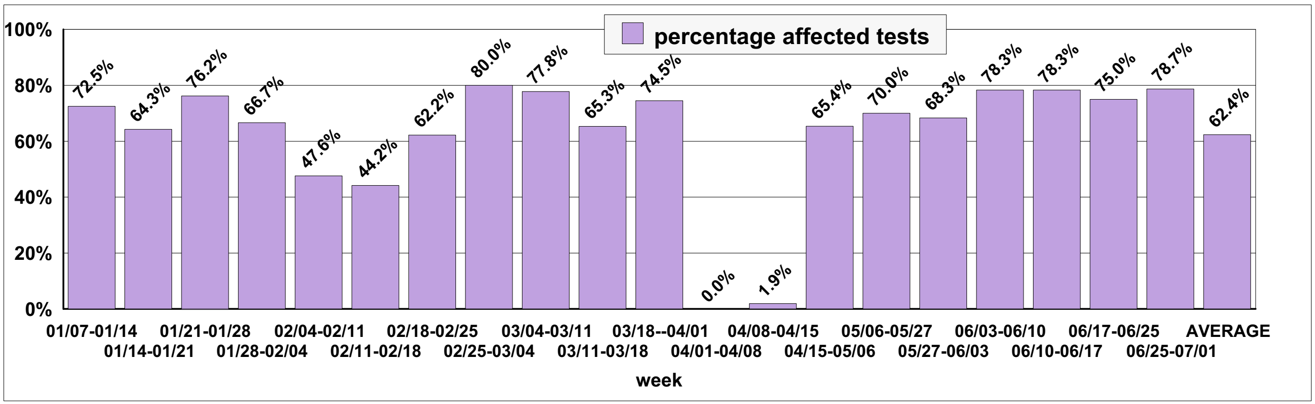

Figure 11. Percentage of affected tests for each of the Daikon versions.

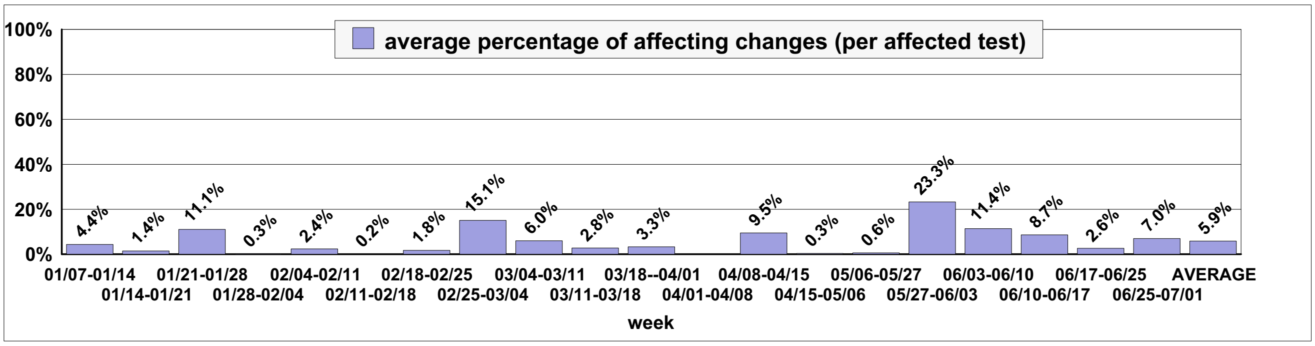

Figure 12. Average percentage of affecting changes, per affected test, for each of the Daikon versions.

tests are affected during the period 04/01-04/08, despite the fact that there were 212 atomic changes during this time. This means that the changed code was not covered by any of the unit 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.

Affecting changes. Figure 12 shows the average percentage of affecting changes per affected test, for each of the Daikon versions. On average, only 5.9% of the atomic changes impacts a given affected test. This means that our techniques have the potential of dramatically reducing the amount of time required for debugging when a test produces an erroneous result after an editing session.

Performance of the tool. The performance of *Chianti* has thus far not been our primary focus, and currently is disappointing. Deriving atomic changes from two versions of Daikon takes approximately 2 minutes[7]. Constructing a 0-CFA call graph for a test takes 10-45 seconds, depending on the amount of code it covers[8]. Generation of XML is currently a performance bottleneck and requires up to 45 seconds per test. Computing the set of affected tests takes 2-3 minutes, and computing affecting changes takes on average about 3 minutes per affected test.

We can think of several concrete steps to significantly improve performance. Determining the affected tests, and the affecting changes for a given test currently involves multiple traversals of the call graph. In principle this is not necessary, and we are working on a redesign that avoids this overhead. Second, faster call graph construction algorithms can be used. For example, XTA [22] scales well and is capable of computing call graphs of 100KLOC programs in a few seconds. However, XTA is less precise than 0-CFA and we need to investigate whether its use significantly degrades precision. Third, the XML generation component was developed in the context of another project, and traverses the call graph multiple times, which is in principle not needed. We expect that reimplementing this component will reduce its running time to a few seconds per call graph. Fourth, one could envision a scenario along the lines of [18], in which call graphs for unit tests are computed "in the background'', while the processor load is low (e.g., during editing).

5. Related Work

We distinguish four broad categories of related work: (i) static techniques that use compile-time information to determine change impact, (ii) dynamic techniques that use run-time information to determine change impact, (iii) regression test selection techniques, and (iv) techniques for understanding and controlling the impact of changes.

Static impact analysis techniques. Recent research on dynamic change impact analyses [13, 12] uses reachability on a call graph as a static technique for comparison. This technique[9] 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 it disregards callers of changed procedures that can also be affected.

Kung *et al.* [10] 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 have some overlap with their class and class library changes.

Tonella's [23] impact analysis 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 slices [8]. Tonella reports some metrics of the computed lattices, but no assessment of the usefulness of his techniques is given.

A number of tools in the Year 2000 analysis domain [5, 14] 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.

Dynamic impact analysis techniques. Orso et. al [13] study impact analysis techniques in order to select and prioritize changes after deployment has taken place. Prior to deployment, applications are instrumented to gather profiling and coverage data at the method/block level. Change impact is determined by correlating a forward static slice [21] w.r.t. a changed program entity with execution data obtained from instrumented applications. The execution data is also used for selecting affected regression tests. There are a number of important differences between our work and that by Orso et al. First, our techniques are intended for use during the earlier stages of software development, to give developers immediate feedback on changes they make. Second, Orso et al. represent changes as sets of methods or blocks that have changed. In contrast, we operate in richer domain of atomic changes with interdependences, which allows us to determine a safe approximation of the changes that may have impacted a given test, a problem not considered by Orso et al. Third, our techniques are based on conservative static analysis, and produce results that hold for any program execution. Techniques based on dynamic

[7] The measurements for call graph construction and XML generation were taken on a Pentium 4 PC at 1.8 Ghz with 1Gb RAM. All other measurements were taken on a Pentium 4 PC at 2.8Ghz with 1Gb RAM.

[8] This call graph construction involves analysis of (large) parts of the standard libraries that are used by the test under consideration, and that must be taken into account in order to compute an accurate call graph.

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

execution data may fail to correctly determine change impact if the executions under consideration do not provide sufficient coverage.

Law and Rothermel [12] present a notion of dynamic impact analysis that is based on whole-path profiling [11]. 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. Since the technique is based on dynamic information, the results of this analysis are only safe w.r.t. the executions under consideration. Thus, their technique might provide a refinement of a safe static analysis such as ours, to show specific impact with respect to a particular program execution of interest.

Zeller [24] 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.

Selective regression testing. Selective regression testing aims at reducing the number of regression tests that must be executed after a software change [15]. These techniques typically determine the units of application code that are covered by a given test, and correlate these against those that have undergone modification to determine tests that are affected. Several notions of coverage have been used. For example, *TestTube* [4] uses a notion of module-level coverage, and *DejaVu* [15] uses a notion of statement-level coverage. The emphasis in this work is mostly on cost reduction, whereas our interest is primarily in assisting maintenance programmers with understanding the impact of program edits.

Bates and Horwitz [1] and Binkley [2] proposed fine-grained notions of 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, for example changes in individual program statements.

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 an order in which to run these tests that maximizes the rate of fault detection. The work is similar to ours in the sense that 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 the non-selection of tests that expose faults.

Other. Palantir [19] 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 [20] 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.

6 Conclusions and Future Work

We have presented our experiences with *Chianti*, a change impact analysis tool that has been validated on 6 months of 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 (62.4%) and for each affected test, the number of affecting changes is very small (5.9% 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.

Our immediate goal is to address the performance issues discussed in Section 4. Plans for future research include experimentation with lower-cost static call graph construction algorithms and with dynamic call graphs, and an in-depth evaluation of the cost/precision tradeoffs involved. Other plans for future work include experimentation with smaller units of change (e.g., basic blocks).

Acknowledgements. We would like to thank Michael Ernst and his research group at MIT for the use of their data.

References

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

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

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

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

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

[6] S. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel, and S. Kanduri. Understanding the effects of changes on the

cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 2003. To appear.

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

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

[9] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. on Programming Languages and Systems*, 23(6):685–746, 2001.

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

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

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

[13] A. Orso, T. Apiwattanapong, and M. J. Harrold. 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.

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

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

[16] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated inclusion constraints. In *Proc. of the ACM SIGPLAN Conf. on Object Oriented Programming Languages and Systems*, pages 43–55, October 2001.

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

[18] D. Saff and M. D. Ernst. Reducing wasted time via continuous testing. In *Proc. of the Fourteenth International Symp. on Software Reliability Engineering*, Denver, CO, November 2003. To appear.

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

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

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

[22] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 281–293, Minneapolis, MN, 2000. *SIGPLAN Notices* 35(10).

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

[24] A. Zeller. 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)*, pages 253–267, Toulouse, France, 1999.