Chianti: A Tool for Change Impact Analysis of Java Programs

Andrej Galad
04/19/2017
CS 6704 - Software Engineering Research
Virginia Polytechnic Institute and State University, Blacksburg, VA
Professor Na Meng
Discussed paper

  - OOPSLA '04 Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications
  - Vancouver, BC, Canada — October 24 - 28, 2004
  - 115 citations
  - It’s long :(
What is change impact analysis?

- Collection of techniques for determining the effects of source code modifications
- **Goals**
  - Experimenting with edits -> *test suite augmentation*
  - Reducing the amount of time spent in regression tests -> *test filtering and selection*
  - Reducing the amount of time spent in debugging -> *fault localization*
Proposed Approach

1. Obtention of interdependent atomic changes
2. Tests’ call graph construction
3. Identification of potentially affected subset of tests
4. Identification of potential behavior affecting changes

- **Previous efforts** - determining subset of the methods in program that were affected by given set of changes
- **Chianti** - isolating subset of the changes that affect given test (via atomic changes)
Overview

- 2 versions of program - affected changes
- Computation of **atomic changes**
  - AC (added class), DC (deleted class), AM (added method), DM (deleted method), CM (changed method), AF (added field), DF (deleted field), LC (lookup change)
  - Syntactic dependencies
    - AM (B.bar()) -> CM (B.bar())
- Generation of call graphs
```java
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();
    }
}
```
Formula

\[
\begin{align*}
\text{AffectedTests}(T, A) &= \{ t_i \mid t_i \in T, \text{Nodes}(P, t_i) \cap (\text{CM} \cup \text{DM})) \neq \emptyset \} \cup \\
&\quad \{ t_i \mid t_i \in T, \; n, A.m \in \text{Nodes}(P, t_i), \\
&\quad \quad n \rightarrow B, X.m \; A.m \in \text{Edges}(P, t_i), \\
&\quad \quad \langle B, X.m \rangle \in \text{LC}, \; B <^* X \} \\
\text{AffectingChanges}(t, A) &= \{ a' \mid a \in \text{Nodes}(P', t) \cap (\text{CM} \cup \text{AM}), \; a' \preceq^* a \} \cup \\
&\quad \{ a' \mid a \equiv \langle B, X.m \rangle \in \text{LC}, \; B <^* X, \\
&\quad \quad n \rightarrow B, X.m \; A.m \in \text{Edges}(P', t), \\
&\quad \quad \text{for some } n, A.m \in \text{Nodes}(P', t), \; a' \preceq^* a \}
\end{align*}
\]

- **Affected test**
  - call graph (in the original version of the program) contains a node that corresponds to a *changed method* (CM) or *deleted method* (DM), or if its call graph contains an edge that corresponds to a *lookup change* (LC).

- **Affecting changes**
  - all atomic changes for *added methods* (AM) and *changed methods* (CM) that correspond to a node in the call graph (in the edited program)
  - atomic changes in the *lookup change* (LC) category that correspond to an edge in the call graph (in the edited program)
  - their transitively prerequisite atomic changes
Extensions for Java

- Initializers, constructors, static fields - AI, DI, CI, ASI, DSI, DSI, CFI, CSFI
- Overloading - CM
- Hierarchy changes - CM
- Exception handling - CM

- How can we match anonymous/local class???

```java
import java.io.*;

class Lister {
    static void listClassFiles(String dir) {
        File f = new File(dir);
        String[] list = f.list();
        for (int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
}

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

class Test {
    static void listJavaFiles(String dir) {
        File f = new File(dir);
        String[] list = f.list();
        for (int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
}
```
Chianti

- Eclipse plugin
  1. Atomic changes derivation (+ call graphs)
  2. Affected tests/changes computation
  3. Visualization
- XML caching
Evaluation

- **Daikon system**
  - Dynamic detection of likely *invariants* (properties that hold at a certain point or points in a program)
  - Source for 2002 - increase from 48K to 123K LOCs
    - Partitions at roughly week intervals
Daikon’s Growth
Change Classification
Evaluation Results
Evaluation Conclusions

● 52% (average) of the tests are affected in **each** edit
  ○ Informs user of sufficient/lacking code coverage

● 3.95% of the atomic changes impact given affected test
  ○ Reduction of debugging time in case of test failures

● Do you find Daikon to be satisfactory evaluation material?
  ○ “Thus, while our change impact analysis findings are promising, they would be more compelling with a test suite offering better coverage of the system”
Performance

- Deriving atomic changes from two successive versions - **87 seconds**
- Computing the set of affected tests for each version pair - **5 seconds**
- Computing affecting changes for given test - **1.2 seconds**

- Do you consider above performance acceptable?
Selective Regression

- Reduction of the number of regression tests required to be executed after a software change
- Affected tests technique
  - Contextless - no need for information regarding test execution
  - Safe - guaranteed to identify any test that reveals a fault
  - Adequacy - sufficiency of test coverage
Related Work

● Change impact analysis
  ○ Bohner and Arnold
    ■ Reachability on a call graph to measure impact (disregards callers)
  ○ CoverageImpact
    ■ Combined methodology (static + dynamic information)
    ■ Program instrumentation
    ■ Association of program entity (block, method) change with a set of possibly affected program entities
  ○ PathImpact
    ■ Dynamic analysis
    ■ Procedure p is changed => any procedure called after p + any procedure on call stack after p returns are potentially impacted

● Regression test selection
  ○ TestTube, DejaVu
  ○ CFG (control flow graphs), JIG (Java interclass graphs)
    ■ Simultaneous traversal of 2 program representations
Future Work

- In-depth evaluation of the cost/precision tradeoffs using static vs. dynamic call graphs
- Smaller units of change
  - Currently - all changes in method $M = CM(M)$
Discussion Question

- How could you make use of *affected tests* results of Chianti?

- How could you make use of *affecting changes* results of Chianti?
Discussion Questions

● How could you improve Chianti’s *evaluation*?

● How could you improve Chianti’s *performance*?
thank you