Helping Programmers Debug Code Using Semantic Change Impact Analysis

Dr. Barbara G. Ryder
Rutgers University
http://www.cs.rutgers.edu/~ryder

This research is supported by NSF grants CCR-0204410 & CCR-0331797 and, in part, by IBM Research. Collaborators are Dr. Frank Tip of IBM T.J. Watson Research Center and graduate students Ophelia Chesley, Xiaoxia Ren, Maximilian Stoerzer and Fenil Shah.

PROLANGS

- Languages/Compilers and Software Engineering
  - Algorithm design and prototyping
- Mature research projects
  - Incremental dataflow analysis, TOPLAS 1988, POPL'88, POPL'90, TSE 1990, ICSE97, ICSE99
  - Pointer analysis of C programs, POPL'91, PLDI'92
  - Side effect analysis of C systems, PLDI'93, TOPLAS 2001
  - *Points-to* and *def-use* analysis of statically-typed object-oriented programs (C++/Java) - POPL'99, OOPSLA'01, ISSTA'02,Invited paper at CC'03, TOSEM 2005, ICSM 2005
  - Profiling by sampling in Java feedback directed optimization, LCPC'00, PLDI'01, OOPSLA'02
  - Analysis of program fragments (i.e., modules, libraries), FSE'99, CC'01, ICSE'03, IEEE-TSE 2004

PROLANGS

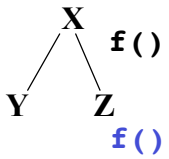
- Ongoing research projects
  - Change impact analysis for object-oriented systems PASTE'01, DCS-TR-533(9/03), OOPSLA'04, ICSM'05, FSE'06, IEEE-TSE 2007
  - Robustness testing of Java web server applications, DSN'01, ISSTA'04, IEEE-TSE 2005, Eclipse Wkshp OOPSLA'05, DCS-TR-579 (7/05)
  - Analyses to aid performance understanding of programs built with frameworks (e.g., Tomcat, Websphere)
  - Using more precise static analyses information in testing OOPLs, DCS-TR-574 (4/05), SCAM'06

Outline

- Overview
- Contributions
- Analysis framework
- Tools built
  - Chianti
    - Experiments with 1 year of Daikon CVS data
  - JUnitCIA
    - Experiments on SE class
  - CRISP
- Related work
- Summary

Non-locality of change impact in OO programs

- Small source code changes can have major and *non-local* effects in object-oriented systems
  - Due to subtyping and dynamic dispatch



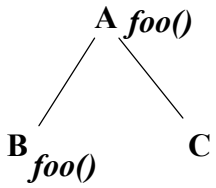

```
X x= new Z()
…
x.f()
```


Motivation

- Program analysis provides feedback on semantic impact of changes
  - E.g., added/deleted method calls, fields and classes
- Object-oriented system presumed to consist of set of classes and set of associated unit or regression tests
- Change impact measured by tests affected
  - Describes application functionality modified
  - Discovers need for new tests

Example

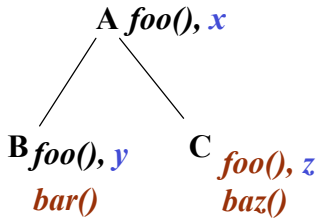
```
class A {
 public void foo() {
}
class B extends A {
 public void foo(){ }
}
class C extends A{
}
```




```
public void test1{
  A a = new A();
  a.foo();//A's foo
}
 public void test2(){
  A a = new B();
  a.foo(); //B's foo
}
public void test3(){
 A a = new C();
 a.foo();//A's foo
}
```


Example

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



Question: what affect did this edit have on these tests?

Example - Changes to foo()

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

```
public void test1{
  A a = new A();
  a.foo();//A's foo
}
 public void test2(){
  A a = new B();
  a.foo(); //B's foo
}
public void test3(){
 A a = new C();
 a.foo();//A's foo
}
```

2.Find affected tests

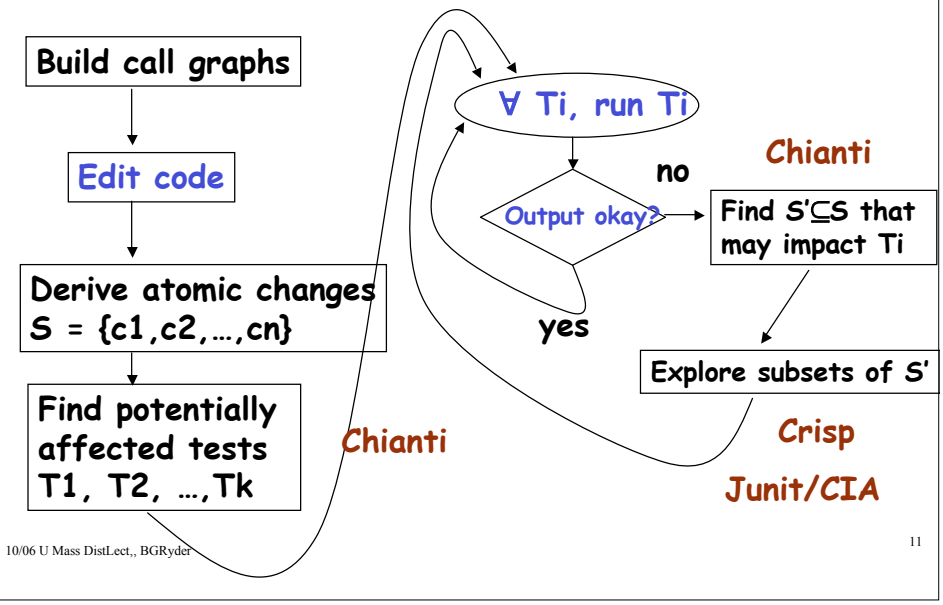
1.Decompose edit into atomic changes

3.Find affecting changes

Assumptions

- Our change analysis tools run within an interactive programming environment - *Eclipse*
  - User makes a program edit
  - User requests change impact analysis of an edit
- Before and after edit, the program compiles
- Tests execute different functionalities of the system
- Call graphs obtainable for each test through static or dynamic analysis
  - Call graphs show the (possible) calling structure of a program; nodes ~ methods, edges ~ calls
- Change impact measured in terms of changes to these call graphs, corresponding to a (growing) set of tests

Usage Scenario



Prototypes

- *Chianti,* a prototype change impact analyzer for full Java language (1.4)
  - Written as an Eclipse plug-in
  - Experimental results from analysis of year 2002 Daikon system CVS check-in
- *Crisp,* a builder of intermediate program versions (between original and edited)
  - To find failure-inducing changes semi-automatically
- *JUnit/CIA,* augmented version of JUnit in Eclipse to perform unit/regression testing and change classification
  - Find likelihood of a change being failure-inducing

Atomic Changes

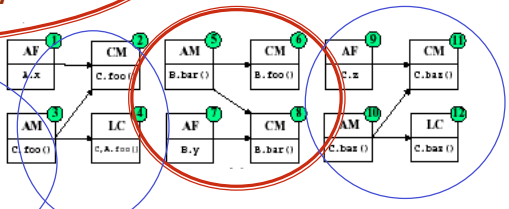
Each edit corresponds to *unique* set of (method-level) atomic changes

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 defn instance field initializer
CSFI Change defn static field initializer
AI Add an empty instance initializer
DI Delete an empty instance initializer
CI Change defn instance initializer
ASI Add empty static initializer
DSI Delete empty static initializer
CSI Change definition of static initializer

Examples of Atomic Changes

```
class A {
 public void foo() { }
 public int x;//1
}
class B extends A {
 public void foo(){B.bar();}//6
 public static void bar()
   {y=17;}//5,8
 public static int y;//7
}
class C extends A{
 public void foo() {
      x = 18; }//2,3
 public void baz() {
      z = 19;}//10,11
 public int z;//9
}
```



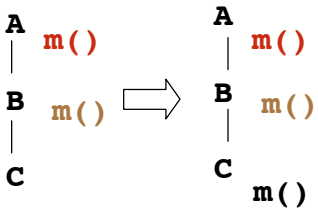
Dynamic Dispatch Changes

A set of triples defined for a class hierarchy:

Runtime receiver type, declared method type signature, actual target method

<C, A.m, B.m> means:

1. class A contains` method m
2. class C does not contain method m
3. class B contains method m, and C inherits m from B



Delete <C, A.m, B.m>
Add <C, A.m, C.m>

LC change: <C, A.m>

Affected Tests

T, set of all tests ; A, set of all atomic changes; P, program before edit; P', program after edit

AffectedTests (T,A) ≡

{Ti | Ti ∈ T, (Nodes(P,Ti) ∩ (CM ∪ DM)) ≠ ∅} ∪

{Ti | Ti ∈ T, n ∈ Nodes(P,Ti), $n \rightarrow_{B,X.m} A.m \in$ Edges(P,Ti), for (B,X.m) ∈ LC, $B < A \leq^* X$}

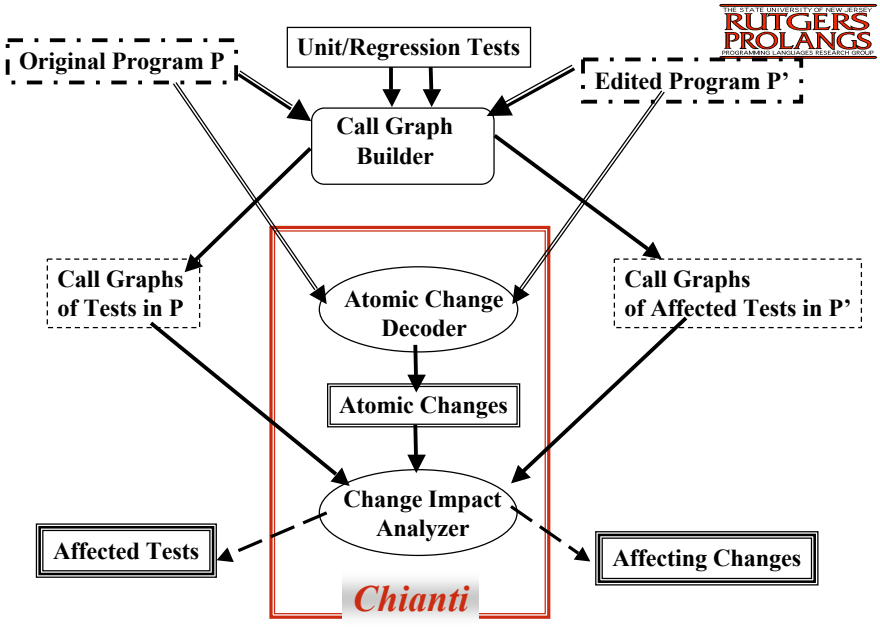
Affecting Changes

T, set of all tests; A, set of all atomic changes; P, program before edit; P', program after edit

Affecting Changes(Ti, A) ≡

{a' | a ∈ Nodes(P',Ti) ∩ (CM ∪ AM), a'<*a} ∪

{a' | a ≡ (B,X.m) ∈ LC, B <* X, $n \rightarrow_{B,X.m}$ A.m ∈ Edges(P',Ti),for some n, A.m ∈ Nodes(P',Ti), a'<*a}



How to run Chianti?

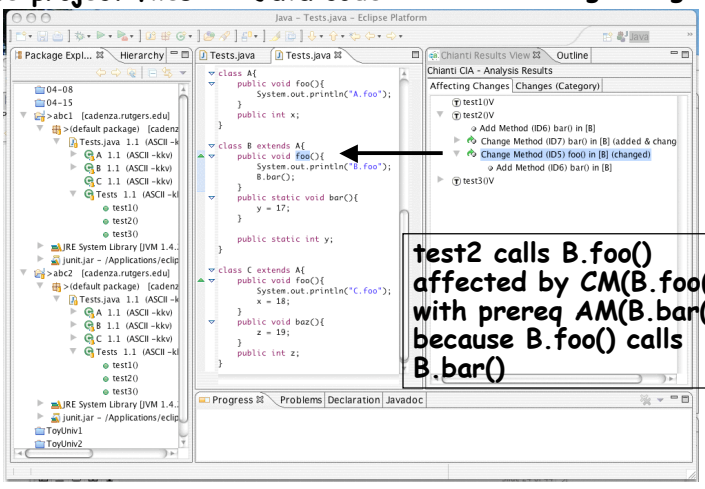
- To simulate use in an IDE while editing evolving code, select 2 successive CVS versions of a project and their associated tests
- Call for change impact analysis of the edit
- Chianti displays change analysis results
  - Affected tests with their affecting atomic changes and prerequisite changes

Chianti GUI

Eclipse project files

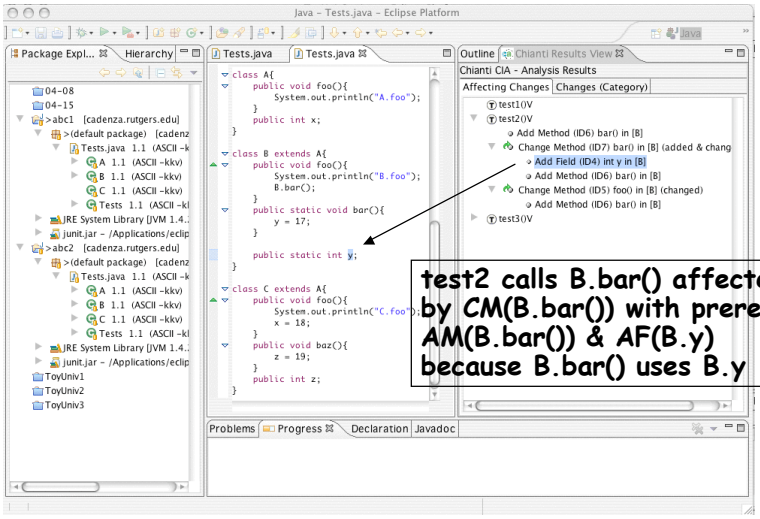
Java code

Affected tests & affecting changes



test2 calls B.foo() affected by CM(B.foo()) with prereq AM(B.bar()), because B.foo() calls B.bar()

Chianti GUI

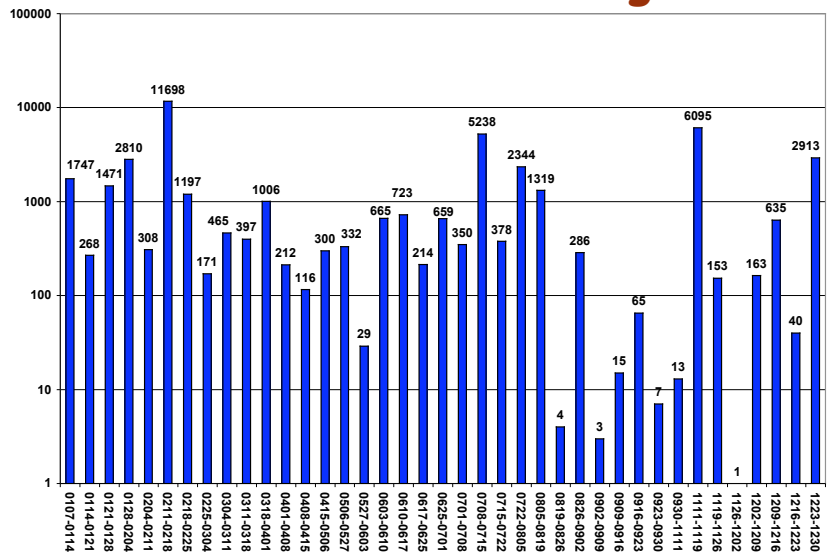


Chianti Experiments

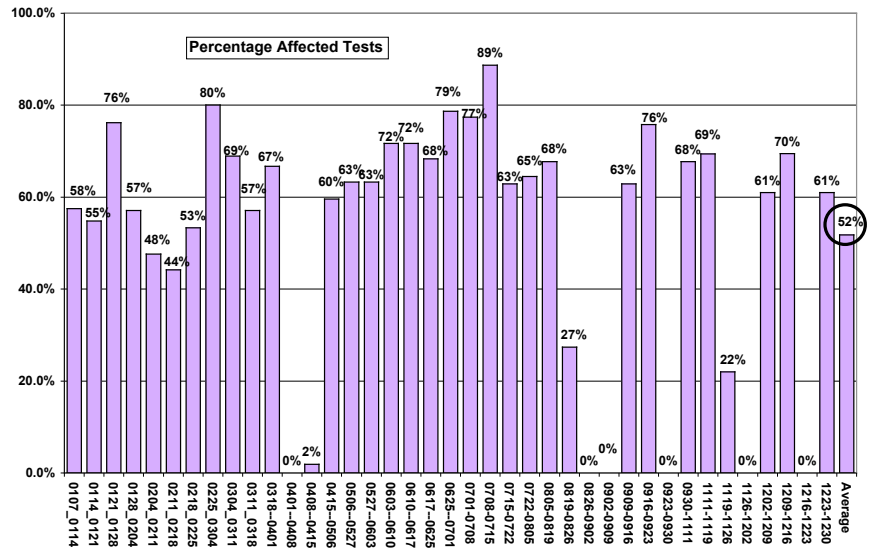
OOPLSA'04

- Data: Daikon project (cf M.Ernst, MIT)
  - Obtained CVS repository from 2002 with version history - an active debugging period
  - Grouped code updates within same week for 12 months
  - Obtained 39 intervals with changes
  - Growth from
    - 48K to 121K lines of code
    - 357 to 765 classes
    - 2409 to 6284 methods
    - 937 to 2885 fields
    - 40-62 unit tests per version

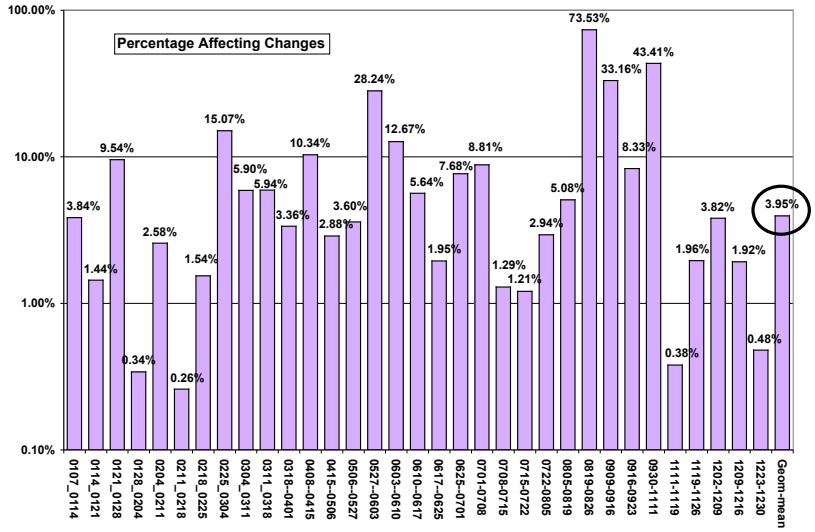
Number of atomic changes



Affected Tests



Affecting Changes



Performance of Chianti

- Deriving atomic changes from 2 successive Daikon versions takes on average 87 seconds
- Calculating the set of affected tests takes on average 5 seconds
- Calculating affecting changes for an affected test takes on average 1.2 seconds
- Results show promise of our change impact framework

FSE'06

JUnit/CIA

- Integrated Chianti with JUnit in *Eclipse*
  - Executes change impact analysis as part of regular unit testing of code during software evolution
    - Allows user to avoid rerunning unaffected unit tests
    - Allows user to see atomic changes not tested by current test suite, in order to develop new tests
- Using test outcome history to estimate likelihood that an affecting change is failure-inducing for a test
  - Classifying changes by whether they are affecting to tests with worsening, improving, or same outcomes on both versions

Junit/CIA - Intuition

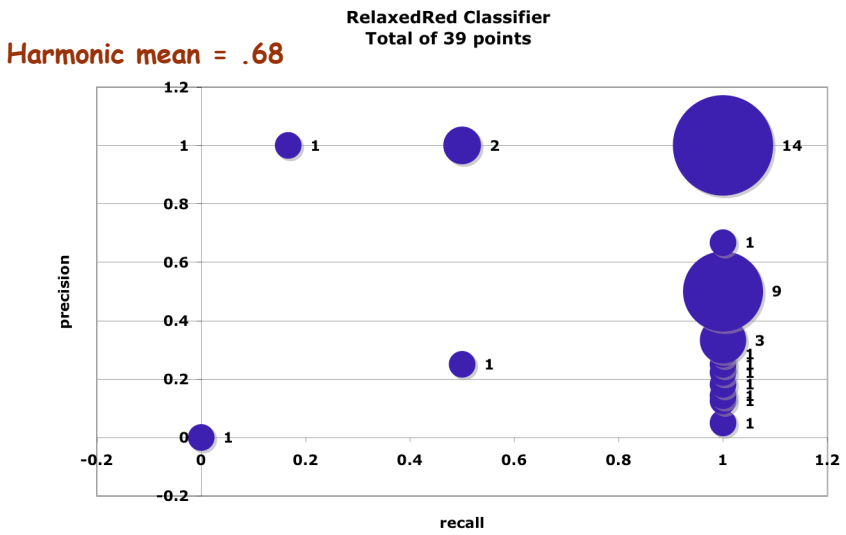
- JUnit/CIA change classifications
  - untested : change affects no test, more tests should be added
  - green: all affected tests improve in outcome
  - yellow: some affected tests worsen, some improve in outcome,indeterminate
  - red: all affected tests worsen in outcome; good candidate for a bug!
- Performed *in situ* experiments with SE class at University of Passau
  - 61 student version pairs had worsening tests with more than 1 affecting change
  - Overall results showed best coloring classifier helped focus programmer attention on failure-inducing changes in ~50% of cases

Comparison of Recall, Precision for Red Classifiers

- Recall
  - Measures fraction of failure-inducing changes colored RED
  - False negative rate = 1-recall
- Precision
  - Ratio of actual failure-inducing RED changes to all RED changes
  - False positive rate = 1-precision
- Pair of values quantifies effectiveness for any RED classifier

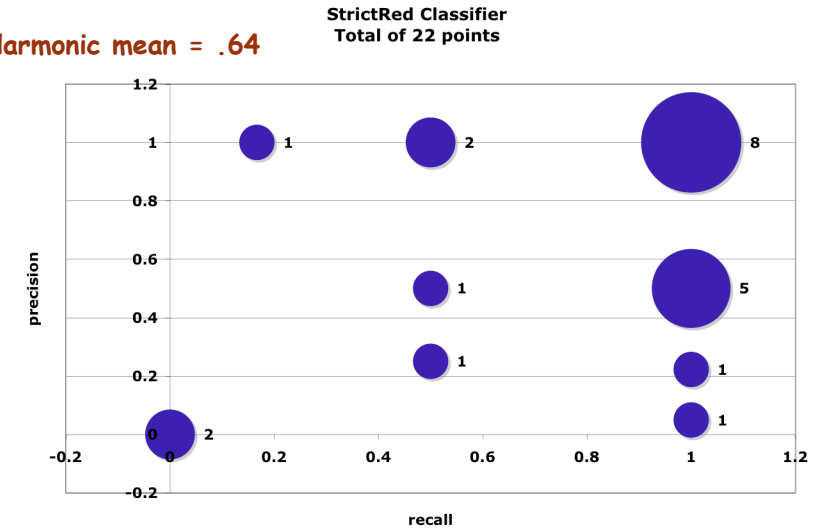
Relaxed Red ("Best") Classifier

Harmonic mean = .68



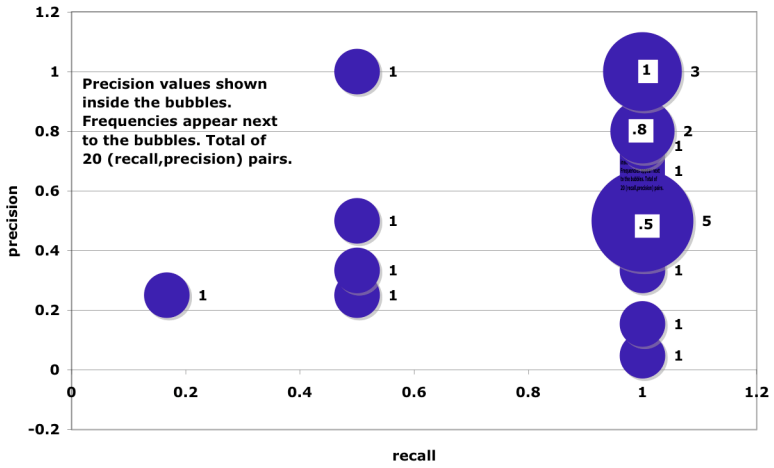
Strict Red Classifier

Harmonic mean = .64



Simple Classifier

Harmonic mean = .49



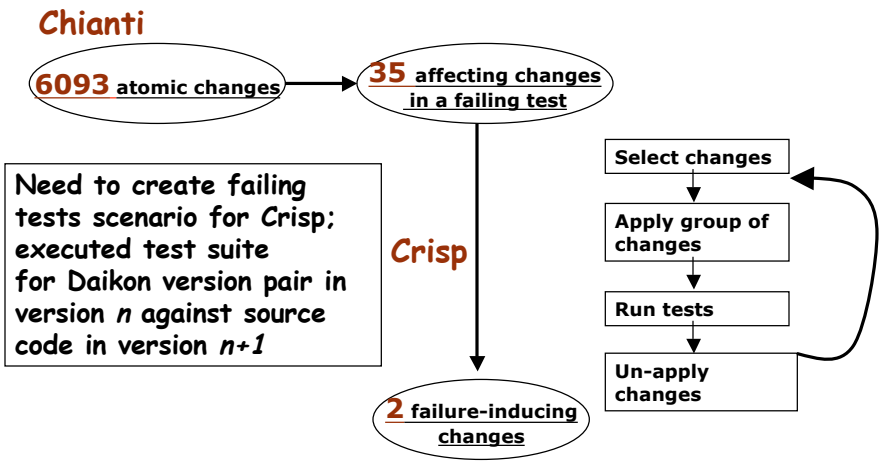
CRISP

ICSM'05, IEEE-TSE 2007

- Tool to find changes introducing bugs
- IDEA: Atomic changes which do not affect a test cannot be responsible for its failure
  - Starts with original program and allows user to select atomic changes to add
  - Tool adds selected changes and their prerequisites to the original version, allowing running of tests on this intermediate program version
    - Goal: to perform this task automatically making 'smart' choices of changes to add

Crisp - Daikon Case Study

Nov 11th (original) and Nov 19th (new) versions



Related Work - Impact Anal

- Static impact analysis techniques
  - Reachability: Bohner-Arnold '96; Kung et al. JSS'96; Tonella TSE 2003;
  - Year 2000 analyses: Eidorff et.al POPL'99; Ramalingam et al. POPL'99;
- Dynamic impact analysis techniques
  - Zeller FSE'99; Law-Rothermel ICSE'03;
- Combined analysis techniques
  - Orso et al. FSE'03; Orso et al. ICSE'04;

Related Work - Testing, Avoiding Recompilation

- Selective regression testing
  - TestTube-Rosenblum ICSE'94; DejaVu-Harrold-Rothermel TOSEM 1997;
  - Slicing-based: Bates-Horwitz POPL'93; Binkley TSE'97;
  - Test selection: Harrold et al. OOPSLA'01; Elbaum et al. JSTVR 2003;
- Techniques for Avoiding Recompilation
  - Tichy ACM TOPLAS'86; et al., ACM TOSEM '94; Hood et al., PSDE'87; Burke et al. TOPLAS'93; Chambers et al., ICSE'95; Dmitriev, OOPSLA'02

Related Work - Fault Localization

- Comparing dynamic data across executions
  - Reps et al. FSE'97; Harrold et al. PASTE'98; Jones et al. ICSE'02; Reneris and Reiss ASE'03; Dallmeier et al. ECOOP'05;
  - Statistically-based techniques: Liblet et al. PLDI'03, PLDI'05; Liu et al. FSE'05;
- Slicing-based techniques
  - Lyle et al. ISICCA'87; DeMillo et al. ISSTA'96; Bonus et al. ASE'03;

Summary

Developed change impact analysis for object-oriented codes with emphasis on practicality, viability in an IDE, and scalability to industrial-strength systems

- Defined atomic changes to subdivide edits
- Defined impact using tests affected and their affecting changes
- Tested ideas in Chianti on 12 months of data from Daikon project with promising results
- Developed new tools --JUnitCIA, CRISP-- that incorporate change impact analysis in practical ways for testing and debugging applications

Future Work

- Upgrade Chianti to Java 1.5
  - Need to consider additional dependences introduced
- Experiment with JUnit/CIA in combination with Crisp to find failure-inducing atomic changes
  - Develop heuristics to automate this process
  - Use larger programs as data -- e.g., Eclipse compiler
- Use change impact analysis to examine collaborative changes to determine unexpected semantic dependences in resulting code
  - Experiment with finding committable changes that allow early exposure of changes to team members
- Explore further applications of change analysis to testing
  - Codify issues of semantic dependence between changes

JUnitCIA Screen Shot