Practical Program Analysis of Object-oriented Software Part 2

Dr. Barbara G. Ryder
Rutgers University

http://www.cs.rutgers.edu/~ryder

September 2006

Outline

Part 2

- New results on accommodating reflection in points-to analysis
- Applications of analysis in software tools to increase programmer productivity
  - Using infeasibility analysis to enable better test coverage for recovery code
    - Testing web server applications for robustness, Fu et. al ISSTA'04, TSE'05, DCS-TR-599
  - Combining static and dynamic analyses to report change impact of edits
    - Change impact analysis for Java codes, OOPSLA'04, FSE'06

Handling Dynamic Class Loading

- New algorithm incrementally accounts for classes loaded and performs analysis updates *online at runtime*
  - Generates constraints at runtime using the JVM and propagates them when a client needs valid points-to results
  - Uses deferred evaluation to handle unresolved references
- Andersen's analysis with field-sensitive object representation, objects represented by their creation sites, and static call graph (CHA)

M.Hirzel, A. Diwan, M. Hind, "Pointer Analysis in the Presence of Dynamic Class Loading", ECOOP 2004

Hirzel et.al Algorithm

- Showed efficacy through use in new connectivity-based GC algorithm
  - Used Jikes RVM 2.2.1 on Specjvm98 benchmarks with good results; claimed need long-running programs for the incremental computation cost to be amortized.
- Validation:
  - Need to make sure points-to solution is updated before do a GC.
  - Then GC verifies the points-to solution by making sure the dynamically observed points-to's are in the solution.

Prolangs Research Project I

Testing robustness of Java web server applications using static and dynamic analyses.

Refs:

C. Fu, B. G. Ryder, "Exception-chain analysis: Revealing Exception Handling Architecture in Java Server Applications", DCS-TR-599, April 2006.

C. Fu, B.G. Ryder, "Navigating Error Recovery Code in Java Applications", *Eclipse Technology Exchange Workshop*, held at *OOPSLA*, October 2005.

C. Fu, A. Milanova, B.G. Ryder, D. Wonnacott, "Robustness Testing of Java Server Applications", *IEEE Transactions on Software Engineering*, Volume 31, Number 4, pp 292-312, April 2005.

C. Fu, B.G. Ryder, A. Milanova, D. Wonnacott, "Testing of Java Web Services for Robustness", in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2004, pp 23-33.

C. Fu, Chen, K. Naragan, T. Nguyen, R. Martin, B.G. Ryder, D. Wonnacott,"Compiler-directed Program Fault Coverage for Highly Available Java Internet Services", in *Proceedings of the International Conference on Dependable Systems and Networks*, June 2003.

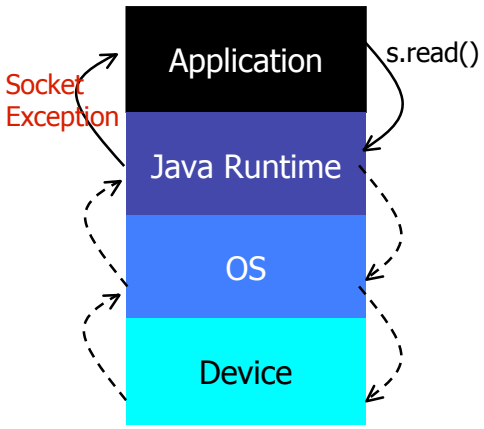
Testing for Robustness

- **Test application reactions to operating system interaction faults**
  - e.g., disk crashes, network congestion, OS resource depletion, OS bugs
- **Provide realistic simulation of fault conditions**
  - Cannot depend on naturally occurring faults
- **Provide measure of robustness of software response**
- **Introduce software analysis and testing methodology to automate this process**

Testing by Fault Injection

Current approach, a form of black box testing:

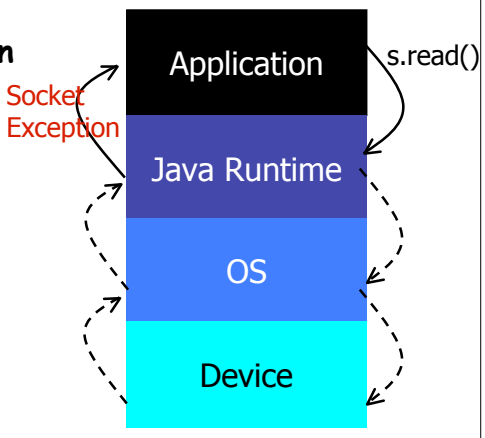
- Learn fault distribution of system by observation
- Inject faults according to observed frequency
- Watch application response: crash/continue
- Measure how often system avoids crash in response to introduced fault



Our Approach

Our approach, a form of white box testing,

- Use knowledge of application from the compiler to inject possibly-affecting faults at appropriate times during execution
- Observe application response (i.e., exception handling)
- Measure coverage of exception handlers



Java Exceptions

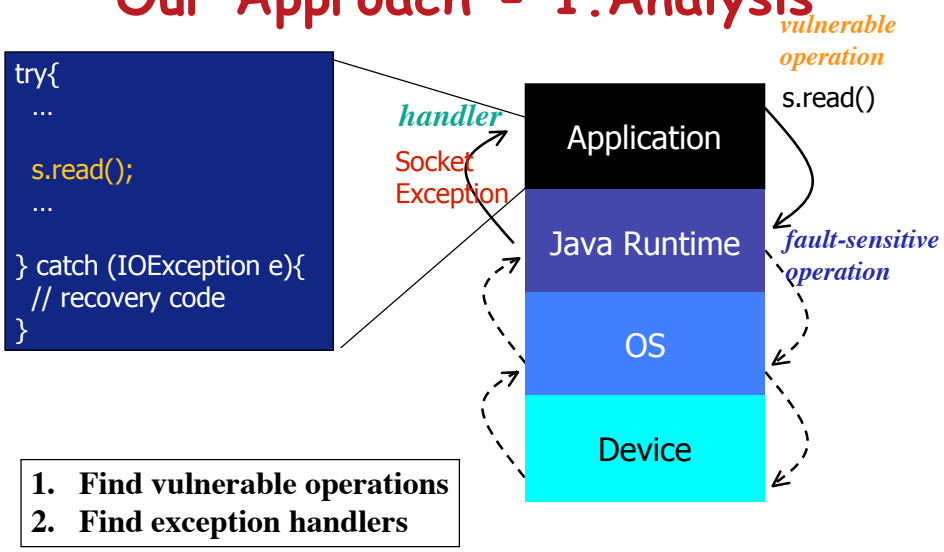
```
void foo() throws Exception{
...
  try{
    bar();
  }
  catch (IOException ioe){…}
}


void bar() throws Exception{
. . .
  throw new SocketException();
. . .
  throw new OtherException();
. . .
}
```

e-c link

OtherException thrown in bar

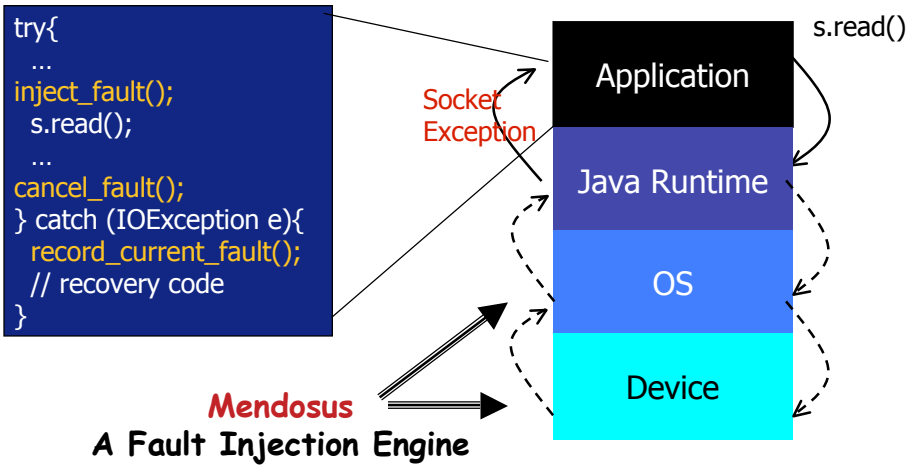
SocketException thrown in bar
OtherException thrown in bar

Our Approach - 1.Analysis

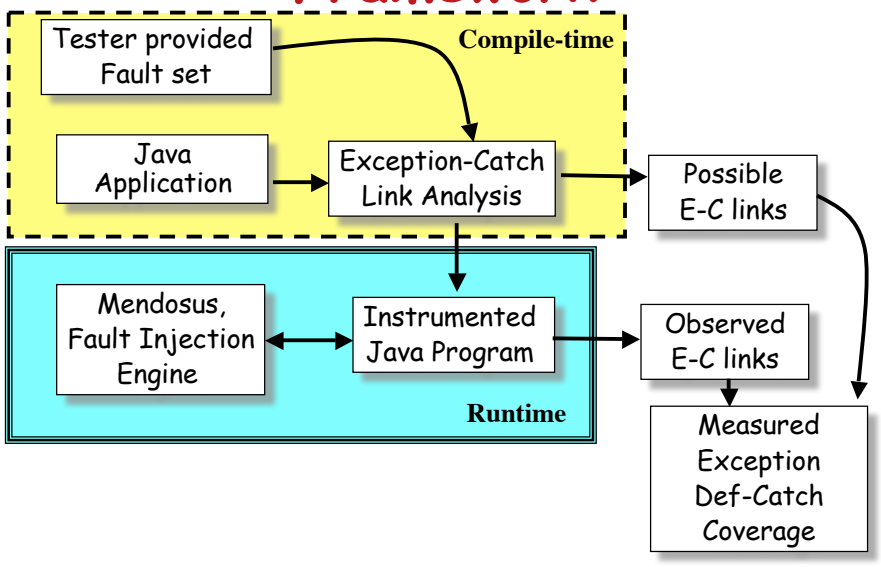


1. Find vulnerable operations
2. Find exception handlers

Our Approach- 2.Instrumentation



Framework



Contributions

- A new white-box testing methodology for recovery code in Java applications
- A practical, fairly precise exception def-use analysis
- A new technique for determining control path infeasibility using data reachability
- A new test coverage metric for fault injection that emphasizes coverage of the application recovery code
- Initial experimental evidence for the practical utility of this approach - achieved on average 84% coverage

Exception-catch Link Analysis

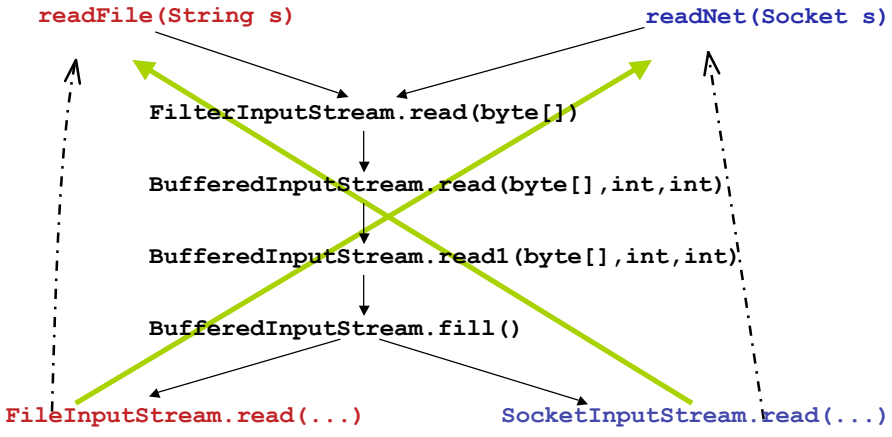
- Two phase algorithm
  - Exception-flow analysis - initial estimate of e-c links using call graph
    - Can vary algm precision by varying precision of call graph construction
  - DataReach analysis - prune away infeasible links using object points-to information
  - Call graph and points-to graphs generated by FieldSens points-to analysis, Rountev et.al OOPSLA'01

DataReach Analysis-Example

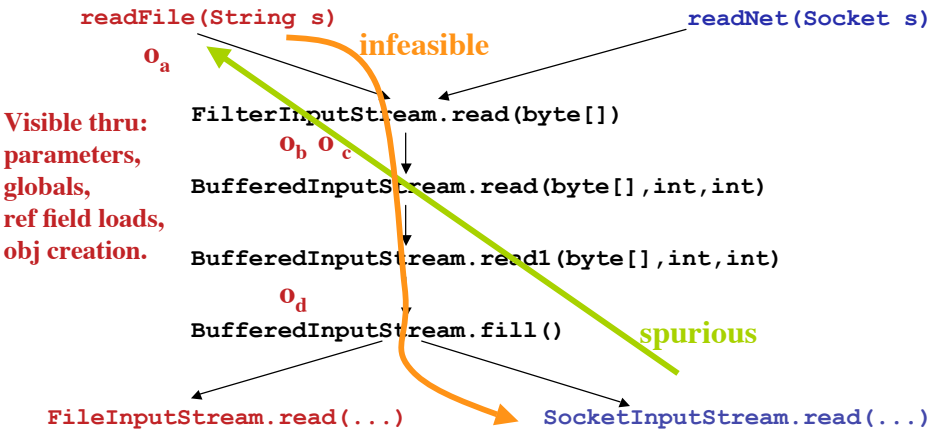
```
void readFile(String s){
 byte[] buffer = new byte[256];
 try{
  InputStream f =new FileInputStream(s);
  InputStream source=new BufferedInputStream(f);
  for (...)
   c = source.read(buffer);
 }catch (IOException e){ ...}
}
```

```
void readNet(Socket s){
 byte[] buffer = new byte[256];
 try{
  InputStream n =s.getInputStream();
  InputStream source=new BufferedInputStream(n);
  for (...)
   c = source.read(buffer);
 }catch (IOException e){ ...}
}
```

Example Call Graph



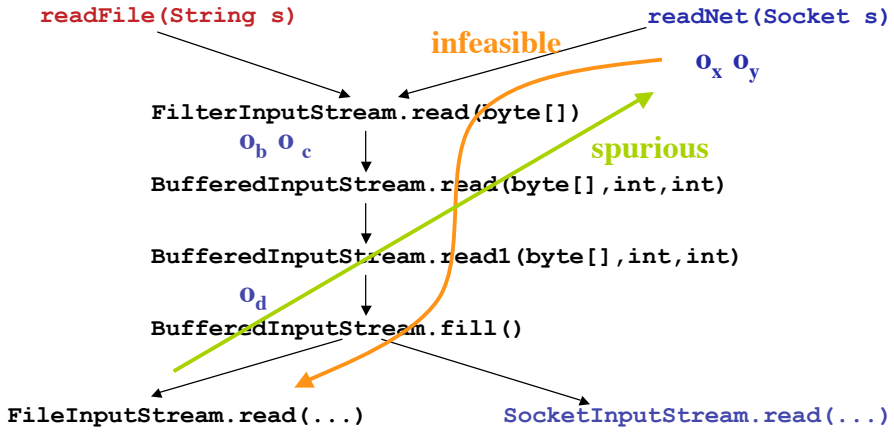
Example Call Graph-2



Visible thru:
parameters,
globals,
ref field loads,
obj creation.

Q: what objects are visible in `BufferedInputStream.fill()` on paths from `readFile()`?

Example Call Graph-3



Q: what objects are visible in `BufferedInputStream.fill()` on paths from `readNet()`?

Experiments

Fu et al, ISSTA'04

- Implemented analysis in McGill's SOOT framework on a 2.8GHz Pentium4 PC under Linux
- Used four moderate-sized benchmarks
- Experimented with different call graph construction algorithms (CHA,RTA,Points-to)
  - Added DataReach separately to see its effect
  - Handled dynamic loading and reflection by dynamic observation
- Focused on recovery code for IOExceptions

Benchmarks

Fu et al, ISSTA'04

Name	*Classes*	*Methods*	*LOC*
FTPD	11(1407)	128(7479)	2783
JNFS	56(1664)	447(9603)	10478
Haboob	338(1403)	1323(7432)	39948
Muffin	278(1365)	2080(7677)	32892

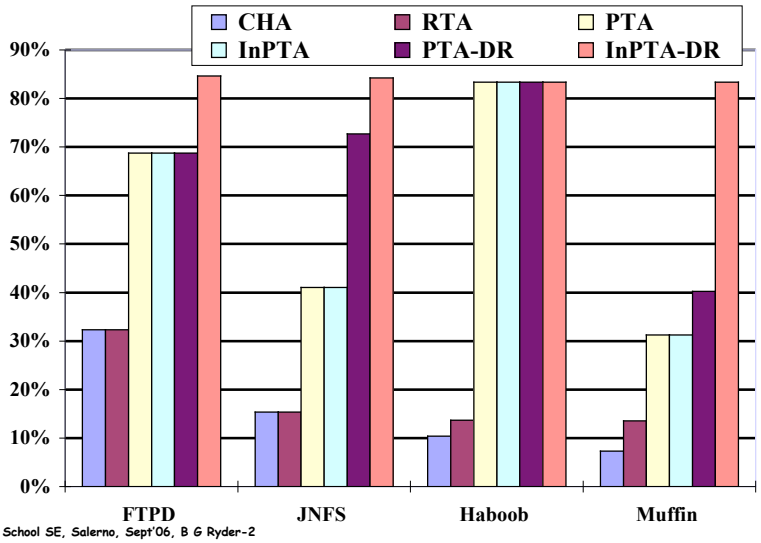
Fu et al, ISSTA'04

Experiment Details

- Analysis combinations tried
  - CHA,Dean et.al, ECOOP'95
  - RTA,Bacon et.al, OOPSLA'96
  - FieldSens points-to,Rountev et.al, OOPSLA'01
  - In-Points-to
  - Points-to/DataReach
  - In-Points-to/DataReach
- Measured coverage as ratio of number of executed e-c links to number of possible e-c links
- Added context sensitivity by inlining constructors that assign to reference fields through the `this` parameter

Fu et al, ISSTA'04

Coverage Data



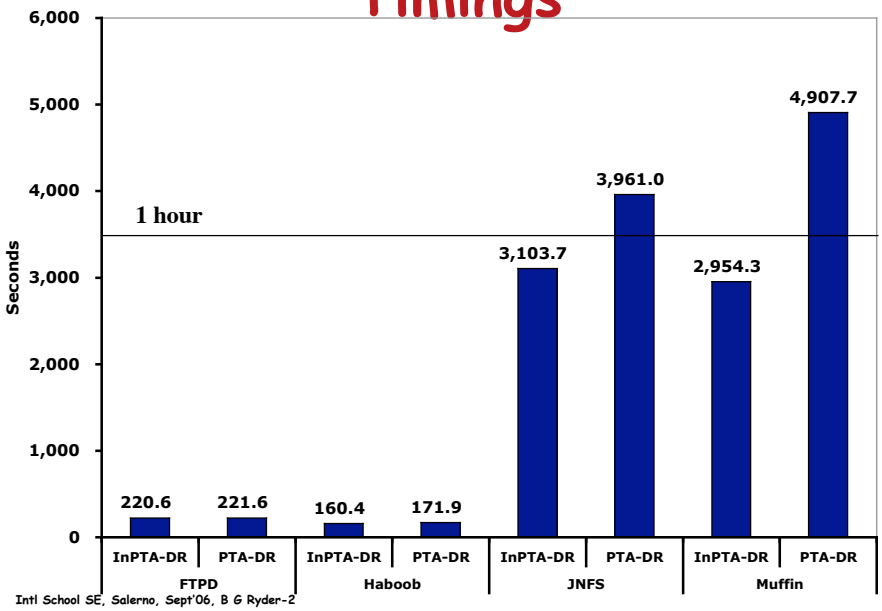
Fu et al, ISSTA'04

Number of e-c links obtained

Program	CHA	In-Points-to/ DataReach	Covered
FTPD	34	13	11
JNFS	104	19	16
Haboob	96	12	10
Muffin	480	42	35

Fu et al, ISSTA'04

Timings



More Recent Results

Fu et al, TSE'05

- Extended the DataReach algorithm
  - M-DataReach: 1 reaching object set per method (implemented)
- Added 3 larger programs to dataset and compared DataReach to M-DataReach on 7 programs
  - HttpClient 1334(4741) methods
  - SpecJVM 2489(4592) methods
  - Vmark 1565(5029) methods
- Added measure *reached* to show e-c links whose try blocks were reached during execution, to calculate *effective coverage*
  - Filter out possible e-c links whose try blocks not *reached*

Recent Results

Fu et al, TSE'05

	InPTA-DR	InPTA-MDR	Effective Coverage
HttpClient	55%	61%	62%
SpecJVM	13%	15%	19%
Vmark	23%	28%	72%

Recent Results

Fu et al, TSE'05

Categorized uncovered e-c links

1. Feasible, but uncovered because of insufficient tests or input
2. Infeasible and difficult to prune for any static analysis
3. Infeasible, but could be eliminated by a more precise static analysis

	1	2	3	Total
Muffin	14%	43%	43%	7
SpecJVM		13%	87%	30
HTTP Cl	25%	60%	15%	40

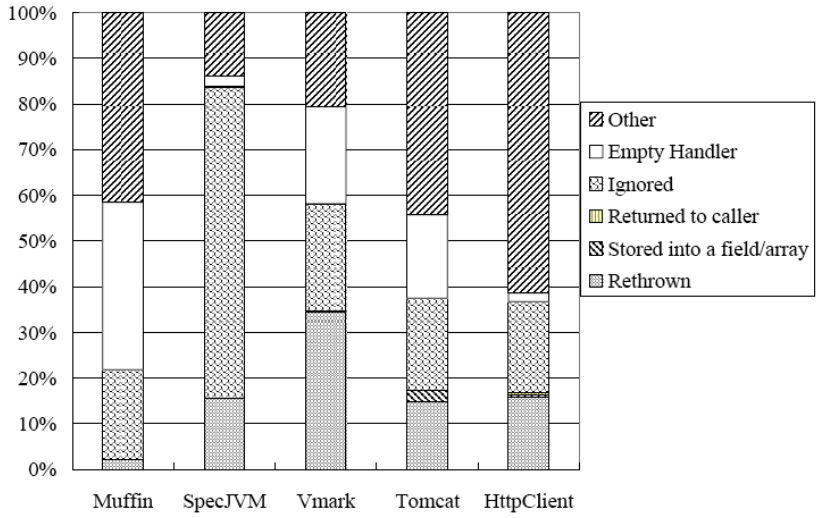
Exception-chain Analysis

- Find exceptions related semantically
  - Rethrows of an exception within a catch that preserves state of entering exception object
  - Enables test and understanding of full recovery code associated with a specific fault
- Also categorized the handlers to show complexity of recovery operations
- Exposes system recovery architecture to user for greater understanding

C. Fu, B. G. Ryder, "Exception-chain analysis: Revealing Exception Handling Architecture in Java Server Applications", DCS-TR-599

Handlers Categorized

Fu et al, DCS-TR-599



Exception-chains Found Starting from a Rethrow

Length	1	2	3	4	5	6	Total
Muffin	6						6
SpecJVM	69	46					115
VMark	300	81	12				393
Tomcat	312	365	31	3	2	10	723
HttpClient	583	547	275				1405

Fu et al, DCS-TR-599

Exception-chain Graph of Tomcat

Fu et al, DCS-TR-599

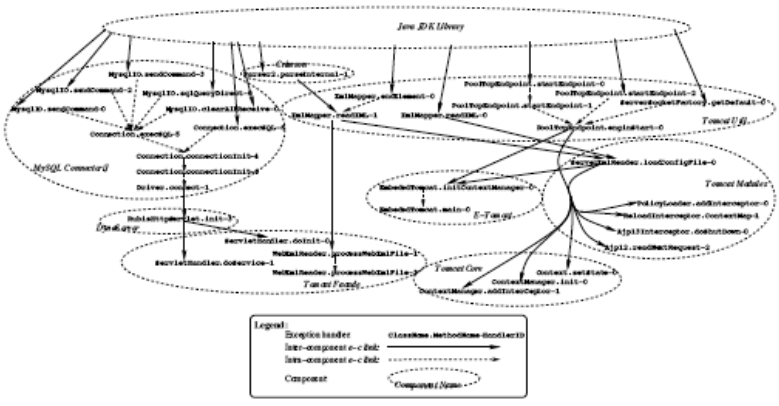


Figure 10: E-c chain Graph of Tomcat

Related Work - Project I

- Dataflow testing and coverage metrics
  - Rapps/Weyuker TSE 4/85; Frankl/Weyuker TSE 10/88,Sinha/Harrold ICSM'99
- Analysis of exception handling
  - Robillard/Murphy TOSEM 2003; Jo et.al JL of Sys and SW 2004; Sinha/Harrold TSE 9/00; Choi et.al PASTE'99;
- Exceptions and Compilation
  - IBM Tokyo JIT, Latte, MRL VM
- Points-to analysis
  - Ryder CC'03; Milanova et. al ISSTA'02; Grove et.al OOPSLA'97; Grove/Chambers TOPLAS 2001; O'Callahan 2000;
- Infeasible paths
  - Bodik et.al FSE'97; Souter/Pollock Info & SW Tech 10/2002;
- Fault injection techniques
  - Arlat et.al IEEE-TSE 8/93;Bouricus et.al NatlConfACM 3/69; Dugan/Trevidi IEEE-TC 6/89
  - Tsai et.al IEEE-TC 11/99;Bieman et.al ISSRE'96

Summary - Project I

- Designed and demonstrated an automated white-box testing methodology for recovery code
- Designed fairly precise def-use analysis for exceptions, including a new technique to eliminate spurious def-uses
- Extended to show exception-handling paths for semantically related exceptions
  - Categorized quality of exception handlers
  - Showed inter-component exception structure
- Promising experimental results with methodology on several Java web applications

Future Work - Project I

- Empirically test exception-chains in Tomcat and other large applications, gathering coverage metrics
  - Learn which chains are hard to cover and why
- Gather exception handler classification data on large, framework-based applications
- Expand set of exceptions tracked by analysis
- Generalize the technique to handle distributed Java applications using RMI

What analyses were used?

- FieldSens points-to analysis augmented with DataReach to show call path infeasibility
- Flow-sensitive def-use analysis for exception objects within a catch block
  - To find rethrows of semantically related exceptions
- Dynamic program instrumentation to record test coverage and record instances of reflection

Prolangs Research Project II

Practical Change Impact Analysis of Java Programs

Refs:

M. Stoerzer, B.G. Ryder, X. Ren, F. Tip, "Finding Failure-Inducing Changes in Java Programs using Change Classification", in the Proceedings of the 14th SIGSOFT Conference on the Foundations of Software Engineering, November 2006.

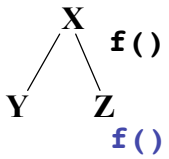
X. Ren, O. Chesley, Ophelia, B.G. Ryder, "CRISP, A Debugging Tool for Java Programs", *IEEE Transactions on Software Engineering*, in press, April 2006.

O. Chesley, X. Ren, B.G. Ryder, "Crisp: A Debugging Tool for Java Programs", in the *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, Budapest, Hungary, September 2005.

X. Ren, F. Shah, F. Tip, B.G. Ryder, O. Chesley, "Chianti: A Tool for Practical Change Impact Analysis of Java Programs", in *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems and Applications (OOPSLA)*, pp 432-448, October 2004.

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



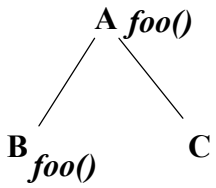

```
X x= new Z()
…
x.f()
```


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

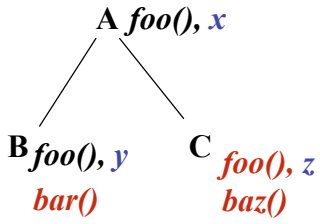
```
class A {
 public void foo() {
}
class B extends A {
 public void foo(){ }
}
class C extends A{
}
```




```
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

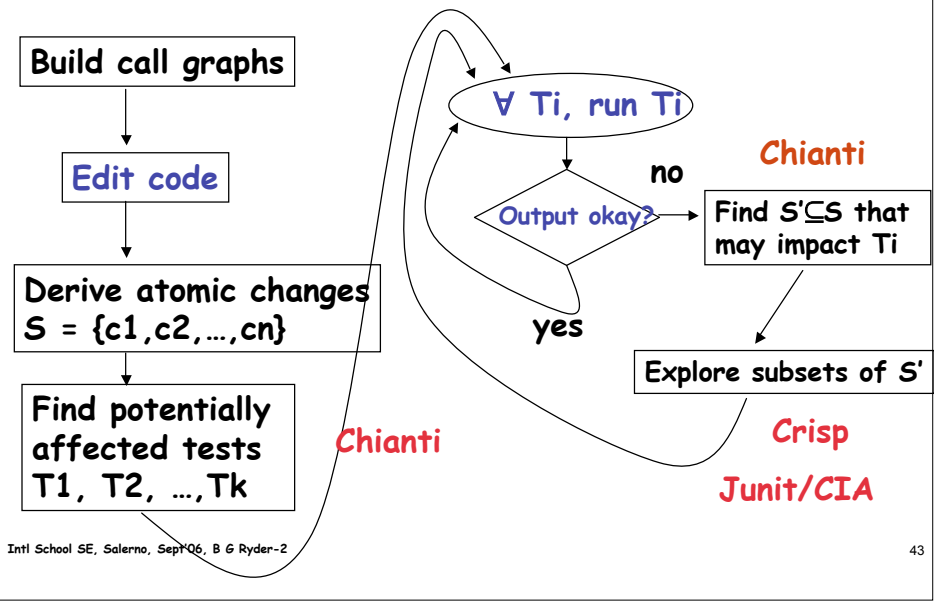
1.Decompose edit into atomic changes

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



Contributions - Tools

- *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

Contributions - Ideas

- Definition of a notion of an atomic change and their interdependences
- Find the set of tests potentially affected by set of atomic changes *Affected Tests*
- For an affected test, find subset of atomic changes that may affect it *Affecting Changes*
- Determine set of atomic changes which do not affect any test
  - Demonstrates need for more tests
- Determine call graph coverage as feedback to aid user in generating new tests

Atomic Changes

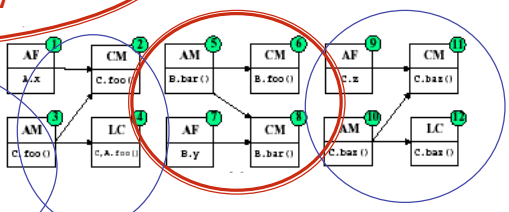
Each edit corresponds to *unique* set of (method-level) atomic changes

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
}
```



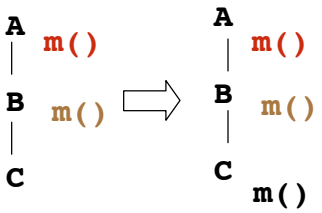
Dynamic Dispatch Changes

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≤* X}

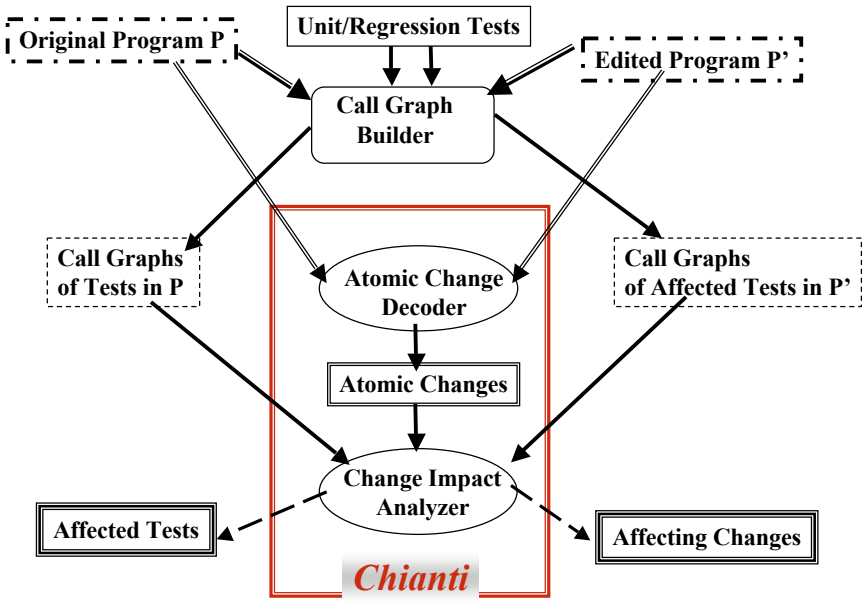
Affecting Changes

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

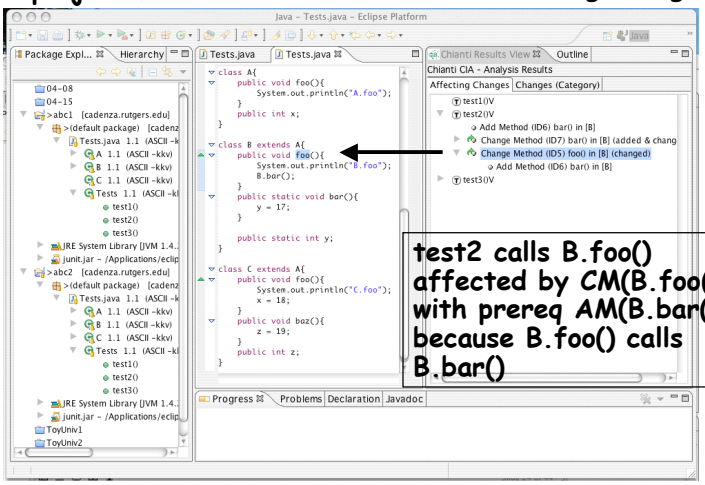


How to run Chianti?

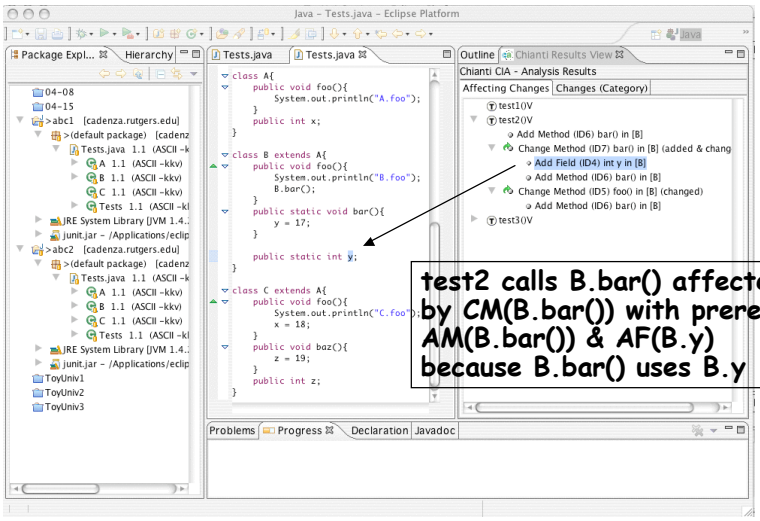
- To simulate use in an IDE while editing evolving code, select 2 successive CVS versions of a project and their associated tests
- Call for change impact analysis of the edit
- Chianti displays change analysis results
  - Affected tests with their affecting atomic changes and prerequisite changes

Chianti GUI

Eclipse project files | Java code | Affected tests & affecting changes



Chianti GUI



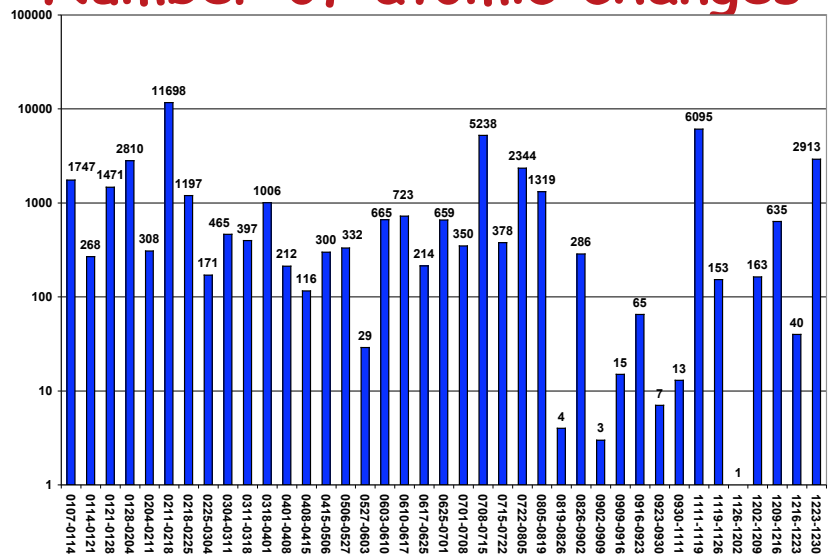
Ren et al, OOPLSA'04

Chianti Experiments

- 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

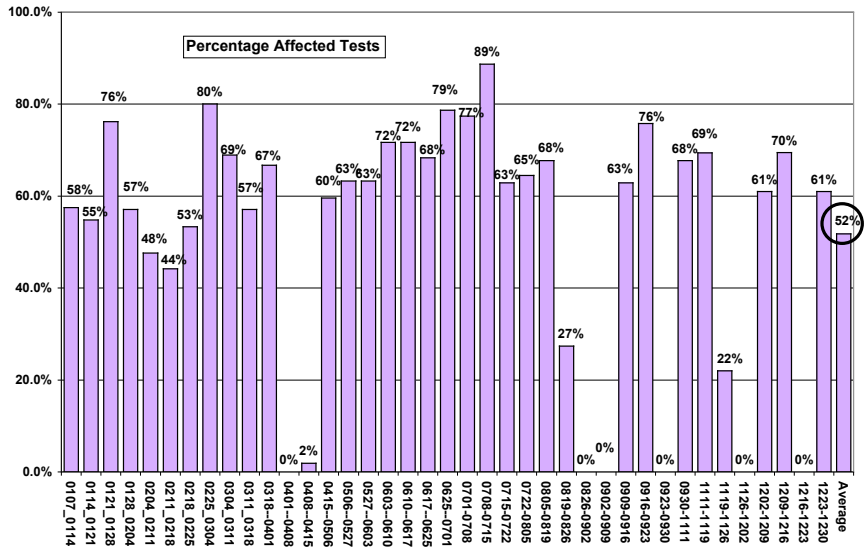
Ren et al, OOPLSA'04

Number of atomic changes



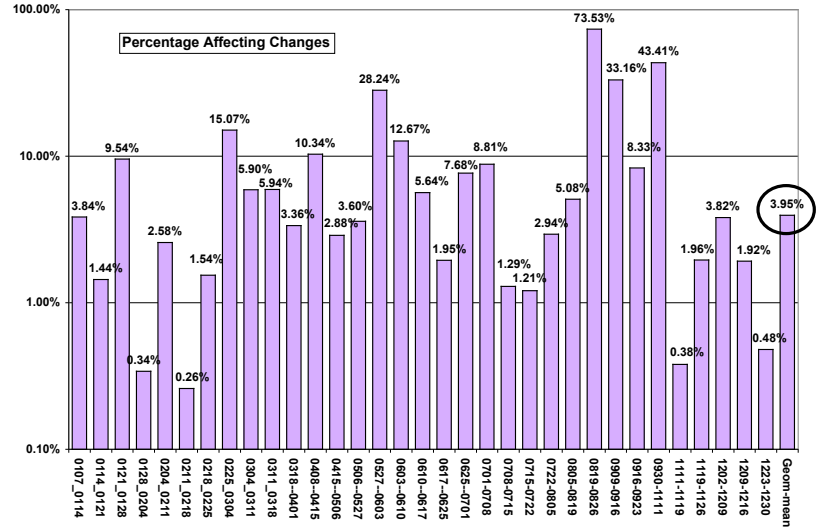
Ren et al, OOPLSA'04

Affected Tests



Ren et al, OOPLSA'04

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

JUnit/CIA

Stoerzer et al, FSE'06

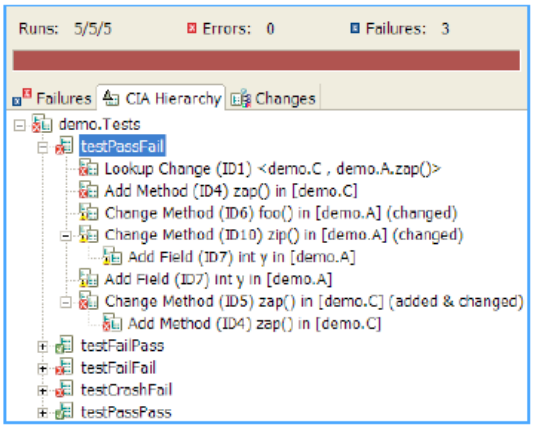
- Integrated Chianti with JUnit in *Eclipse*
  - Executes change impact analysis as part of regular unit testing of code during software evolution
    - Allows user to avoid rerunning unaffected unit tests
    - Allows user to see atomic changes not tested by current test suite, in order to develop new 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

Stoerzer et al, FSE'06

- 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
  - Overall results showed best (coloring) classifier helped focus programmer attention on failure-inducing changes in ~50% of cases

JUnitCIA Screen Shot



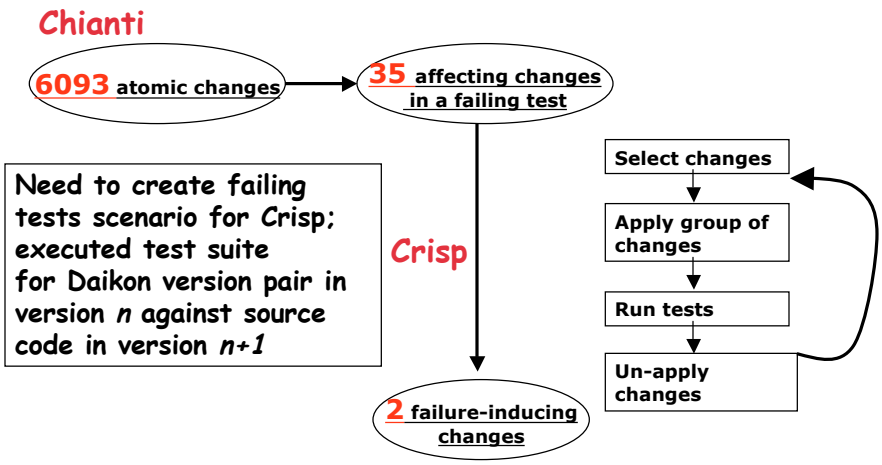
Chesley et al ICSM'05,
Ren et al, TSE'06 inpress

CRISP

- 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 - Project II

Impact Analysis

- 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 - Project II

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 - Project II

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 - Project II

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 - Project II

- Experiment in Crisp with heuristics to suggest an exploration order of the affecting changes to the user
  - Comparison of several possible heuristics for adding in affecting changes to build new intermediate program versions
  - Automation of the running of the test on the new intermediate program version
  - Try technique on larger programs -- e.g., Eclipse compiler
- Use change impact analysis to examine collaborative changes to determine unexpected semantic dependences in resulting code

What analyses were used?

- Combines dynamic analysis (e.g., test call graphs) with static analysis to find Affected tests and their affecting changes
  - Also can be used with call graphs constructed using FieldSens points-to
- Use dynamic analysis to run tests, obtain outcomes and measure test coverage

References

- O. Agesen. *The cartesian product algorithm: Simple and precise type inference of parametric polymorphism.* In European Conference on Object Oriented Programming, pages 2–26, 1995
- L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, 1994. Also available as DIKU report 94/19.
- D. Bacon and P. Sweeney, *Fast Static Analysis of C++ Virtual Function Calls, OOPSLA'96*
- M.Berndl, O. Lhotak, F. Qian, L. Hendren, N.Umanee, *Points-to analysis using BDDs*, PLDI'03, pp 103-114.
- R. Chatterjee. *Modular Data-flow Analysis of Statically Typed Object-oriented Programming Languages*. PhD thesis, Department of Computer Science, Rutgers University, October 1999.
- R. Chatterjee, B. G. Ryder, and W. Landi. *Relevant context inference*. In Symposium on Principles of Programming Languages, pages 133–146, 1999.
- J. Dean, D. Grove, C. Chambers, *Optimization of OO Programs Using Static Class Hierarchy,* ECOOP'95

References

- D. Grove and C. Chambers. *A framework for call graph construction algorithms.* ACM Transactions on Programming Languages and Systems (TOPLAS), 23(6), 2001.
- M. Hirzel, A. Diwan, M. Hind, *Pointer Analysis in the Presence of Dynamic Class Loading,* ECOOP 2004, pp 96-122.
- O. Lhotak and L. Hendren, *Scaling Java Points-to Analysis using Spark*, In Proceedings of the Twelvth International Conference on Compiler Construction, Warsaw, Poland, April 2003.
- O. Lhotak and L. Hendren, *Jedd: a BDD-based relational extension of Java,* PLDI'04, pp 158-169.
- D. Liang, M. Pennings, and M.J. Harrold. *Evaluating the precision of static reference analysis using profiling.* In Proceedings of the international symposium on software testing and analysis (ISSTA), pages 22–32. 2002
- D. Liang, M. Pennings, M.J. Harrold, *Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs,* Workshop on Program Analysis for Software Tools and Engineering, (PASTE), pp 6-12, 2006.

References

- A. Milanova, A. Rountev, and B.G. Ryder. *Parameterized object-sensitivity for points-to and side-effect analyses for Java*. In International Symposium on Software Testing and Analysis (ISSTA), pages 1–11, 2002.
- A. Milanova, A. Rountev, B.G. Ryder, *Parameterized Object Sensitivity for Points-to Analysis for Java,* in ACM Transactions on Software Engineering Methodology, Volume 14, Number 1, pp 1-41, January 2005.
- M. Naik, A. Aiken, J. Whaley, *Effective static race detection for Java*, PLDI'06, pp 308-319.
- N. Oxhoj, J. Palsberg, and M. Schwartzbach. *Making type inference practical*. In European Conference on Object-Oriented Programming, pages 329–349, 1992.
- J. Palsberg and M. Schwartzbach. *Object-oriented type inference*. In Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 146–161, 1991

References

- J. Plevyak and A. Chien. *Precise concrete type inference for object oriented languages.* In Proceeding of Conference on Object-Oriented Programming Systems, Languages and Applications.(OOPSLA '94), pages 324–340, October 1994.
- A. Rountev, A. Milanova, B.G. Ryder, *Points-to Analysis for Java Using Annotated Constraints*, OOPSLA'01
- E. Ruf. *Effective synchronization removal for Java*. In Conference on Programming Language Design and Implementation, pages 208–218, 2000.
- B.G. Ryder, *Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages*, invited paper in the Proceedings of the Twelveth International Conference on Compiler Construction, Warsaw, Poland, April 2003, pp 126-137.
- B. Steensgaard. *Points-to analysis in almost linear time*. In Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, pages 32–41, 1996.

References

- M. Sridharan and R. Bodik, *Refinement-based context-sensitive points-to analysis for Java,* PLDI'06, pp3 87-400.
- V.Sundaresan, et. al, Practical Virtual Method Call Resolution for Java, OOPSLA'00
- F. Tip and J. Palsberg, *Scalable Propagation-based Call Graph Construction Algorithms,* OOPSLA'00
- J. Whaley and M. Lam. *An efficient inclusion-based points-to analysis for strictly-typed languages.* In Static Analysis Symposium, 2002.
- J. Whaley and M. Lam, *Cloning-based context-sensitive pointer alias analysis using binary decision diagrams*, PLDI'04, pp 131-144.
- J. Zhu and S. Calman, *Symbolic pointer analysis revisited*, PLDI'04, pp 145-157.