Reducing the Costs of Bounded-Exhaustive Testing

Vilas Jagannath, Yun Young Lee, Brett Daniel
and Darko Marinov

University of Illinois at Urbana-Champaign

FASE '09 - York, UK, 22 - 29 March, 2009

Bounded-Exhaustive Testing

Automated testing approach that checks a system under test for all inputs within given bounds

- Many faults can be revealed with small inputs
- Exhaustive testing within bounds catches "corner cases"

Used in academia and industry to test real-world applications

- Refactoring Engines - Eclipse & NetBeans [Daniel et al. FSE 07]
- Web Traversal Agent from Google [Misailovic et al. FSE 07]
- XPath Compiler at Microsoft [Stobie ENTCS 05]
- Constraint Analyzer [Khurshid & Marinov J-ASE 04]
- Fault-Tree Analyzer for NASA [Sullivan et al. ISSTA 04]
- Protocol for Dynamic Networks [Khurshid & Marinov ENTCS 01]

Steps of Bounded-Exhaustive Testing

User

- Describes inputs and bounds
- Provides test oracles

Tool

- Generates all inputs within bounds
- Executes them on system under test
- Checks outputs using oracles

User

- Waits for generation/execution/checking
- Inspects failing tests

Costs of Bounded-Exhaustive Testing

User . . . . Human time

- Describes inputs and bounds
- Provides test oracles

Tool . . . . Machine time

- Generates all inputs within bounds
- Executes them on system under test
- Checks outputs using oracles

User . . . . Human time

- Waits for generation/execution/checking
- Inspects failing tests

Costs can be significant

Example magnitudes from our case study

- 1-2 hours to describe inputs (not addressed in this paper)
- Thousands of inputs generated/executed/checked
- Total testing time takes hours
- Finding the first failure can take tens of minutes
- Hundreds of failing tests need to be inspected

Contributions: Reducing several costs

Three novel techniques that reduce several costs

- Machine time
- Human waiting time
- Inspection effort

Case study: Testing of Eclipse Refactoring Engine

Refactorings & Refactoring Engines

Refactorings are behavior-preserving program transformations that improve program design

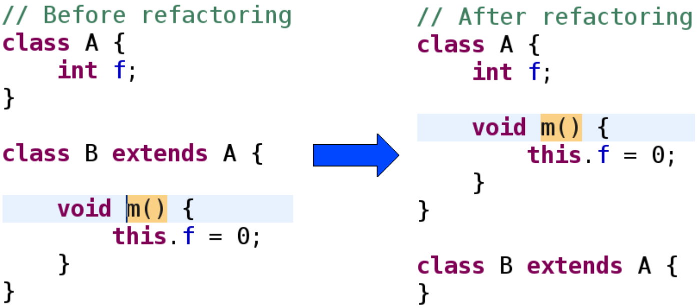
- Change internals of code, not external behavior
- Examples: rename class, move method, encapsulate field, etc.

Refactoring engines are tools that automate the application of refactorings

- Key component of most modern IDEs such as Eclipse

Refactoring Example: Pull Up Method

Moves a method from a subclass into one of its superclasses



```java
// Before refactoring
class A {
    int f;
}

class B extends A {

    void m() {
        this.f = 0;
    }
}
```

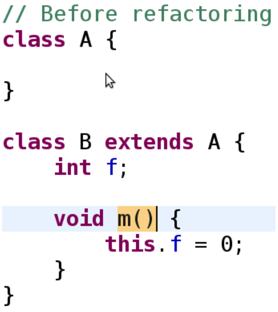
```java
// After refactoring
class A {
    int f;

    void m() {
        this.f = 0;
    }
}

class B extends A {
}
```

Refactoring Example: Pull Up Method

Moves a method from a subclass into one of its superclasses



```
// Before refactoring
class A {

}

class B extends A {
    int f;

    void m() {
        this.f = 0;
    }
}
```

Warning: Cannot move ‘m’ without moving ‘f’

Why Test Refactoring Engines?

Widely used

Complex

- Complex inputs: programs
- Complex code: program analysis and transformation

Can silently corrupt large parts of programs

- A bug in refactoring engine can be as unpleasant as a bug in compiler or libraries

Bounded-Exhaustive Testing of Refactoring Engines

ASTGen framework [Daniel et al. FSE 07]:

- Allows users to write Abstract Syntax Tree (AST) generators
- Provides library of basic generators which can be composed
- Executes generators to generate ASTs (all within bounds)
- Applies refactorings on generated ASTs
- Checks results with oracles

ASTGen: Example Inputs

Description: Three classes related through sub/super class and inner/outer class relationships. A sub class has a method that refers to a field in a super class and also has another method that invokes that method

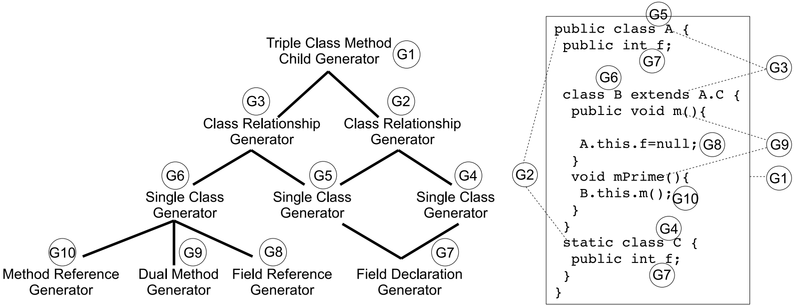
```
public class A {
 public int f;
 class C {
  public int f;
 }
}
class B extends A.C {
  private void m(){
   this.f=0;
  }
  void mPrime(){
   m();
  }
}
```

```
public class A {
 public int f;
 class B extends C {
  private void m(){
   new A().f=0;
  }
  void mPrime(){
   m();
  }
 }
}
class C {
 public int f;
}
```

```
public class A {
 public int f;
 class B extends C {
  private void m(){
   super.f=0;
  }
  void mPrime(){
   m();
  }
 }
 class C {
  public int f;
 }
}
```

ASTGen: Example Generator

Triple Class Method Child Generator:



Triple Class Method Child Generator

Class Relationship Generator

Class Relationship Generator

Single Class Generator

Single Class Generator

Single Class Generator

Method Reference Generator

Dual Method Generator

Field Reference Generator

Field Declaration Generator

```
public class A {
 public int f;

 class B extends A.C {
  public void m(){

   A.this.f=null;
  }
  void mPrime(){
   B.this.m();
  }
 }
 static class C {
  public int f;
 }
}
```

Results and Costs

Promising results

- Dozens of faults found and reported in Eclipse and NetBeans
- Being included in the NetBeans testing process

Costs

Refactoring	**Generator**	**Num of Inputs**	**Total Time**	**TTFF**	**Num of Failures**	**Num of Faults**
EncapsulateField	DualClassFieldReference	23760	427:09	73:34	486	3
PullUpMethod	TripleClassMethodChild	1152	27:02	9:09	160	2
	DualClassMethodChild	576	13:22	n/a	0	0
RenameField	DualClassFieldReference	23760	629:01	n/a	0	0

Time To First Failure (TTFF)

- User wait time after starting tool until a failing test is found
- Important metric in an interactive testing scenario

Three Techniques to Reduce Costs

Sparse Test Generation

- Reduces TTFF (but increases the total time)

Structural Test Merging

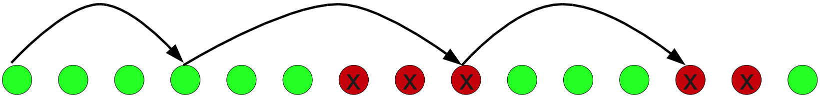
- Reduces the total time and TTFF

Oracle-based Test Clustering

- Reduces human effort for inspection

Sparse Test Generation

Observation: Failing tests often located close together due to combinatorial nature of generation



Intuition: Jump through input space to find failures faster

- Width and periodicity of failing runs unknown, so random jumps within bounded length

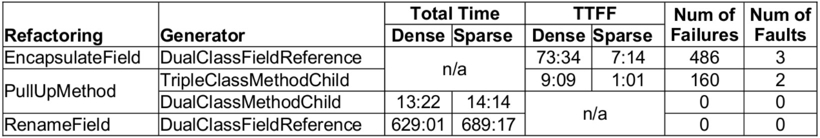
Sparse Test Generation

Two passes through test generation:

- Sparse Generation
    - Jumps through the generation sequence with random jumps within bounded length
    - Significantly improve TTFF while slightly increasing total time
    - Random jump lengths between 1-20, expect ∼10% increase in total time
- Exhaustive Generation
    - Performs basic exhaustive generation
    - No compromise in failure-detection capability

Sparse Test Generation Results

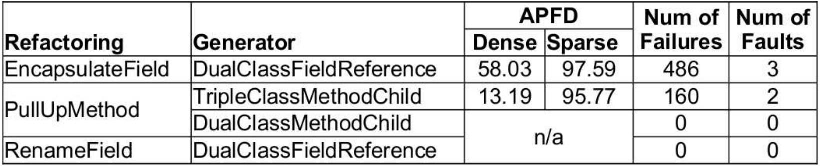
- Up to 10x improvement in TTFF
- ∼10% increase in Total Time



Refactoring	Generator	Total Time		TTFF		Num of Failures	Num of Faults
		Dense	Sparse	Dense	Sparse		
EncapsulateField	DualClassFieldReference	n/a		73:34	7:14	486	3
PullUpMethod	TripleClassMethodChild			9:09	1:01	160	2
	DualClassMethodChild	13:22	14:14	n/a		0	0
RenameField	DualClassFieldReference	629:01	689:17			0	0

Sparse Test Generation Results

- Also significantly improves APFD (Average Percentage Fault Detection)



Refactoring	Generator	APFD		Num of Failures	Num of Faults
		Dense	Sparse		
EncapsulateField	DualClassFieldReference	58.03	97.59	486	3
PullUpMethod	TripleClassMethodChild	13.19	95.77	160	2
	DualClassMethodChild	n/a		0	0
RenameField	DualClassFieldReference			0	0

Structural Test Merging

Inspired by previous work on Test Granularity [Rothermel et al. ICSE 02]

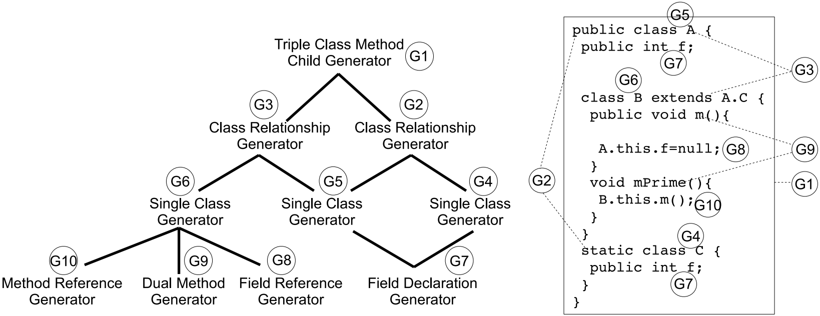
- Append smaller tests to form larger tests
- Smaller number of larger tests rather than larger number of smaller tests
- Save setup and teardown costs
- Could mask old faults or reveal new faults

Challenge and solution

- Cannot generally append two ASTs to form larger ASTs
- Merge structurally smaller inputs to form larger inputs
- Save setup, teardown, and execution costs

Structural Test Merging: Recall the Example

Unmerged generator structure:



```
public class A {
 public int f;

 class B extends A.C {
  public void m(){

   A.this.f=null;
  }
  void mPrime(){
   B.this.m();
  }
 }
 static class C {
  public int f;
 }
}
```

Structural Test Merging: Unmerged inputs

Unmerged test inputs from generator

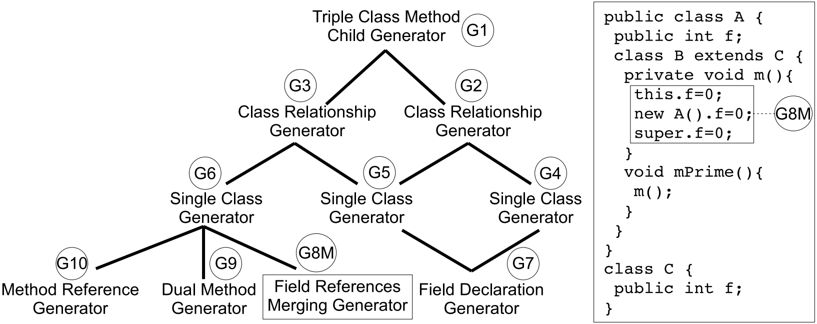
```
public class A {
 public int f;
 class B extends C {
  private void m(){
   this.f=0;
  }
  void mPrime(){
   m();
  }
 }
}
class C {
 public int f;
}
```

```
public class A {
 public int f;
 class B extends C {
  private void m(){
   new A().f=0;
  }
  void mPrime(){
   m();
  }
 }
}
class C {
 public int f;
}
```

```
public class A {
 public int f;
 class B extends C {
  private void m(){
   super.f=0;
  }
  void mPrime(){
   m();
  }
 }
}
class C {
 public int f;
}
```

Structural Test Merging: Merged versions

Merged generator and test input



Triple Class Method Child Generator G1

G3 Class Relationship Generator

G2 Class Relationship Generator

G6 Single Class Generator

G5 Single Class Generator

G4 Single Class Generator

G10 Method Reference Generator

G9 Dual Method Generator

G8M Field References Merging Generator

G7 Field Declaration Generator

```
public class A {
 public int f;
 class B extends C {
  private void m(){
   this.f=0;
   new A().f=0;
   super.f=0;
  }
  void mPrime(){
   m();
  }
 }
}
class C {
 public int f;
}
```

Structural Test Merging: Results

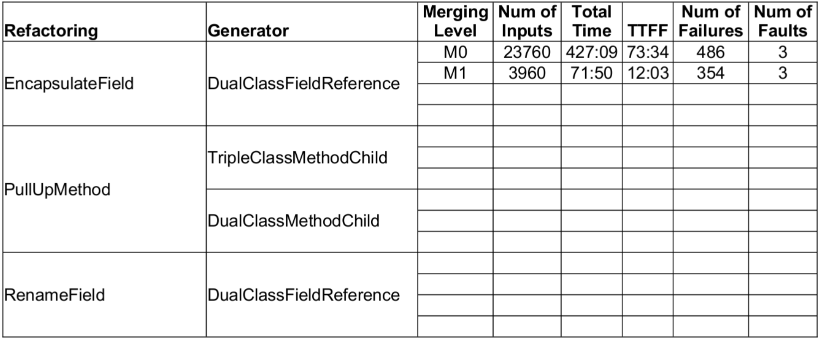
Orders of magnitude reduction in total time
No reduction in fault detection for M1 (but not always for higher)

Refactoring	**Generator**	**Merging Level**	**Num of Inputs**	**Total Time**	**TTFF**	**Num of Failures**	**Num of Faults**
EncapsulateField	DualClassFieldReference	M0	23760	427:09	73:34	486	3
PullUpMethod	TripleClassMethodChild						
	DualClassMethodChild						
RenameField	DualClassFieldReference						

Structural Test Merging: Results

Orders of magnitude reduction in total time

No reduction in fault detection for M1 (but not always for higher)



Refactoring	Generator	Merging Level	Num of Inputs	Total Time	TTFF	Num of Failures	Num of Faults
EncapsulateField	DualClassFieldReference	M0	23760	427:09	73:34	486	3
		M1	3960	71:50	12:03	354	3
PullUpMethod	TripleClassMethodChild						
	DualClassMethodChild						
RenameField	DualClassFieldReference						

Structural Test Merging: Results

Orders of magnitude reduction in total time

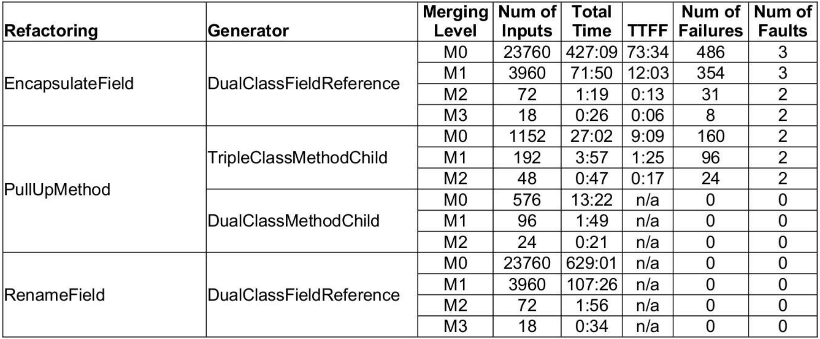
No reduction in fault detection for M1 (but not always for higher)

Refactoring	**Generator**	**Merging Level**	**Num of Inputs**	**Total Time**	**TTFF**	**Num of Failures**	**Num of Faults**
EncapsulateField	DualClassFieldReference	M0	23760	427:09	73:34	486	3
		M1	3960	71:50	12:03	354	3
		M2	72	1:19	0:13	31	2
		M3	18	0:26	0:06	8	2
PullUpMethod	TripleClassMethodChild						
	DualClassMethodChild						
RenameField	DualClassFieldReference						

Structural Test Merging: Results

Orders of magnitude reduction in total time

No reduction in fault detection for M1 (but not always for higher)



Refactoring	Generator	Merging Level	Num of Inputs	Total Time	TTFF	Num of Failures	Num of Faults
EncapsulateField	DualClassFieldReference	M0	23760	427:09	73:34	486	3
		M1	3960	71:50	12:03	354	3
		M2	72	1:19	0:13	31	2
		M3	18	0:26	0:06	8	2
PullUpMethod	TripleClassMethodChild	M0	1152	27:02	9:09	160	2
		M1	192	3:57	1:25	96	2
		M2	48	0:47	0:17	24	2
	DualClassMethodChild	M0	576	13:22	n/a	0	0
		M1	96	1:49	n/a	0	0
		M2	24	0:21	n/a	0	0
RenameField	DualClassFieldReference	M0	23760	629:01	n/a	0	0
		M1	3960	107:26	n/a	0	0
		M2	72	1:56	n/a	0	0
		M3	18	0:34	n/a	0	0

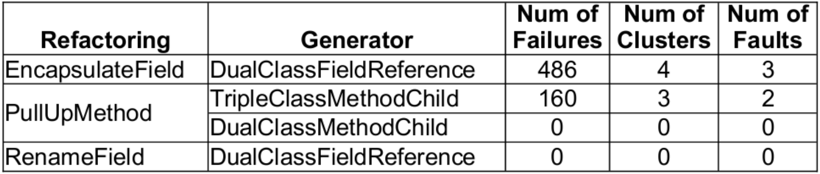
Oracle-based Test Clustering

Inspired by work in test clustering/filtering/indexing/bucketing

- Relies on oracles that provide more than just pass/fail
- Groups failing tests based on oracle information to reduce the inspection time
- Abstraction of information provided by oracles
    - “field f not visible” instead of “field f not visible at line 2 col 5”

Oracle-based Test Clustering Results

Handful of clusters instead of hundreds of failures



Refactoring	Generator	Num of Failures	Num of Clusters	Num of Faults
EncapsulateField	DualClassFieldReference	486	4	3
PullUpMethod	TripleClassMethodChild	160	3	2
	DualClassMethodChild	0	0	0
RenameField	DualClassFieldReference	0	0	0

Comparison with three other techniques available in the paper

Conclusions

Bounded-Exhaustive Testing effective but has many costs
Presented three techniques that reduce some costs

- Sparse Test Generation reduces TTFF
- Structural Test Merging reduces total machine time
- Oracle-based Test Clustering reduces human inspection effort

Ongoing work: reduce human effort in writing generators

- UDITA: unified declarative/imperative generation
- Promising results: shorter generators (easier to write), faster generation, more bugs