Advanced Program Analyses for Object-oriented Systems

Dr. Barbara G. Ryder
Rutgers University
http://www.cs.rutgers.edu/~ryder
http://prolangs.rutgers.edu/
July 2007

Lecture 5 - Outline

• Uses of analysis in software tools
  - For performance diagnosis, program understanding, slicing and testing
  - Idea: choose an appropriate analysis for a problem
  - Projects:
    1. Testing recovery code in Java programs
    2. Accurate interclass test dependence to guide integration testing
    3. Blended (static & dynamic) analysis for performance diagnosis

Thanks to my graduate students: Bruno Dufour, Chen Fu and Weilei Zhang for help with these slides.
Testing Robustness of Java SW

Idea

• Use static analysis to find exception handling paths in Java program fairly accurately;

• Use fault injection engine to test exception handling through communication with instrumented Java program

• Replace current black box testing approach

C. Fu, B.G. Ryder, “Exception-chain Analysis: Revealing Exception Handling Architecture in Java Server Applications”, ICSE’07;

Testing by Fault Injection

The approach, a form of white box testing, uses knowledge of application from the compiler to inject possibly-affecting faults and record their handling
Our Approach - 1. Analysis

```java
try{
    ...
    s.read();
    ...
} catch (IOException e){
    // recovery code
}
```

```
Application
```

```
Java Runtime
```

```
OS
```

```
Device
```

```
Socket Exception
```

```
vulnerable operation
```

```
s.read()
```

```
fault-sensitive operation
```

Our Approach - 2. Instrumentation

```java
try{
    ...
    inject_fault();
    s.read();
    ...
    cancel_fault();
} catch (IOException e){
    record_current_fault();
    // recovery code
}
```

```
Application
```

```
Java Runtime
```

```
OS
```

```
Device
```

```
Mendosus Fault Injection Engine
```

```
s.read()
```

```
Socket Exception
```

```
fault-sensitive operation
```

```
s.read()
```

```
vulnerable operation
```
Exception-catch Link Analysis

- Two phase algorithm
  - Exception-flow analysis - initial estimate of e-c links
  - DataReach analysis - prune away links corresponding to infeasible call paths, using points-to information

- Need fairly accurate call graph to trace exception handling up the call stack
  - Techniques - CHA, RTA, FieldSens - determine precision of analysis
  - Whole-program dataflow analysis
    - Backward propagation on call graph from exception throws to catch block handlers
    - Intraprocedural propagation maintains ordering between try blocks; partially flow-sensitive, context-insensitive
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){ ...}
}
**DataReach Analysis**

![Diagram showing the relationship between readFile(String s), readNet(Socket s), FilterInputStream.read(byte[]), and BufferedInputStream.read(byte[], int, int).]

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

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

**Experiment Benchmarks**

<table>
<thead>
<tr>
<th>Name</th>
<th>Classes</th>
<th>Methods</th>
<th>Try Blocks</th>
<th>.class Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTPD</td>
<td>11(1407)</td>
<td>128(7479)</td>
<td>17</td>
<td>39,218</td>
</tr>
<tr>
<td>JNFS</td>
<td>56(1664)</td>
<td>447(9603)</td>
<td>36</td>
<td>175,297</td>
</tr>
<tr>
<td>Muffin</td>
<td>278(1365)</td>
<td>2080(7677)</td>
<td>270</td>
<td>727,118</td>
</tr>
<tr>
<td>Haboob</td>
<td>338(1403)</td>
<td>1323(7432)</td>
<td>134</td>
<td>731,413</td>
</tr>
<tr>
<td>HttpClient</td>
<td>252(2210)</td>
<td>1334(4741)</td>
<td>536</td>
<td>1,049,784</td>
</tr>
<tr>
<td>SpecJVM</td>
<td>484(2161)</td>
<td>2489(4592)</td>
<td>219</td>
<td>2,817,687</td>
</tr>
<tr>
<td>VMark</td>
<td>307(2266)</td>
<td>1565(5029)</td>
<td>502</td>
<td>2,902,947</td>
</tr>
</tbody>
</table>

*C. Fu et al, TSE'05*
Experiment Details

- Analysis combinations tried:
  CHA, RTA, FieldSens, In-Points-to, Points-to/DataReach, In-Points-to/DataReach, In-Points-to/MD ataReach

- Measured coverage as ratio of number of executed e-c links to number of possible e-c links

- Added some context sensitivity by inlining constructors that assign to reference fields through the this parameter

---

E-C Link Coverage Data A

C. Fu et al, TSE'05
Uncovered E-C Links

- 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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Muffin</td>
<td>1(14%)</td>
<td>3(43%)</td>
<td>3(43%)</td>
<td>7</td>
</tr>
<tr>
<td>SpecJVM</td>
<td></td>
<td>4(13%)</td>
<td>26(87%)</td>
<td>30</td>
</tr>
<tr>
<td>HTTPClient</td>
<td>10(25%)</td>
<td>24(60%)</td>
<td>6(15%)</td>
<td>40</td>
</tr>
</tbody>
</table>
Exception-Catch Chains

- Semantically-related exceptions
  - Preserve state of original exception object in new exception throw, or
  - Re-throw original exception object
  - Called re-thrown exceptions
- E-c chains of re-thrown exceptions
  - Discover through flow-sensitive analysis of each catch clause
  - Link together exception handling paths of re-thrown exceptions to find e-c chains

E-C Chains found

<table>
<thead>
<tr>
<th>Lengths: Prgms:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Muffin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>SpecJM</td>
<td>69</td>
<td>46</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>115</td>
</tr>
<tr>
<td>Vmark</td>
<td>300</td>
<td>81</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>393</td>
</tr>
<tr>
<td>Tomcat</td>
<td>312</td>
<td>365</td>
<td>31</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>723</td>
</tr>
<tr>
<td>HttpClient</td>
<td>583</td>
<td>547</td>
<td>275</td>
<td></td>
<td></td>
<td></td>
<td>1405</td>
</tr>
</tbody>
</table>
Improving Class-based Testing

- Class testing for OOPLs is unit testing
- Classes can be interdependent imposing an order on how best to integrate and test them
  - Dependence cycles can exist -- solve by stubs or big-bang
- Deriving more accurate Interclass Test Dependence (ICTD) through code analysis can help
  - May allow parallelization of integration test; more testing in same time period
  - ICTD also can be useful for program understanding and software visualization

W. Zhang & B.G. Ryder, “Discovering Accurate Interclass Test Dependences”, PASTE’07
ORD-Based Definition

- Causes for ICTD:
  - Inheritance
  - Aggregation (part-of)
  - Association (uses)
  - Polymorphism

B is test dependent on A, C and D

Transitive closure of the above
Sometimes there are spurious cycles, and associations which are not semantic dependences at runtime

Motivating Example

```java
class DeliveryHandler{
    public void handleDelivery(DeliveryTrans deliveryTrans){
        deliveryTrans.process();
        deliveryTrans.display(outFile);
    }
}
```

Red arrow because handleDelivery calls DeliveryTrans methods process() and display() but they have no influence on the execution of their caller
Blue arrow because there is a valid dependence of DeliveryTrans on DeliveryHandler
Semantics-based Definition of ICTD

- There is test dependence from class $A$ to class $B$, if there is a statement $s$ in a method callable on an $A$ object and a statement $t$ in a method declared in $B$, such that:
  - $s$ may have visible side effects (i.e., $s$ may either write to the external memory or return a value), and
  - $s$ is semantically dependent on $t$ while testing class $A$

Zhang & Ryder, PASTE'07

Key Ideas

- Approximate at method-level granularity for scalability
- Three causes for method dependence:
  - Caller uses return value of callee
  - Callee is control-dependent on caller
  - Side effect: one method writes to the same memory region that another method reads
    - Requires reference analysis to identify region
- Propagate dependences on the call graph

Zhang & Ryder, PASTE'07
Analysis Configurations

- Algorithm parameterized by choice of analyses to calculate the call graph and side effects
- Four analysis configuration applied:
  - VTA: variable type analysis (call graph not constructed on the fly)
  - 0CFA: 0-CFA (call graph constructed on the fly)
  - OB: 1-object-sensitive points-to analysis
  - OBR: OB+R (i.e., DataReach)

Zhang & Ryder, PASTE'07

ICTD Reduction Rate wrt. ORD-based Definition

<table>
<thead>
<tr>
<th>Library</th>
<th>VTA</th>
<th>0CFA</th>
<th>OB</th>
<th>OBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>41.28%</td>
<td>62.42%</td>
<td>10.62%</td>
<td>50.00%</td>
</tr>
<tr>
<td>jpeg</td>
<td>33.33%</td>
<td>54.54%</td>
<td>8.88%</td>
<td>53.33%</td>
</tr>
<tr>
<td>raytrace</td>
<td>41.28%</td>
<td>62.42%</td>
<td>10.62%</td>
<td>50.00%</td>
</tr>
<tr>
<td>db</td>
<td>41.28%</td>
<td>62.42%</td>
<td>10.62%</td>
<td>50.00%</td>
</tr>
<tr>
<td>j Cec</td>
<td>41.28%</td>
<td>62.42%</td>
<td>10.62%</td>
<td>50.00%</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>41.28%</td>
<td>62.42%</td>
<td>10.62%</td>
<td>50.00%</td>
</tr>
<tr>
<td>mrt</td>
<td>41.28%</td>
<td>62.42%</td>
<td>10.62%</td>
<td>50.00%</td>
</tr>
<tr>
<td>jack</td>
<td>41.28%</td>
<td>62.42%</td>
<td>10.62%</td>
<td>50.00%</td>
</tr>
<tr>
<td>jbb</td>
<td>41.28%</td>
<td>62.42%</td>
<td>10.62%</td>
<td>50.00%</td>
</tr>
<tr>
<td>Average</td>
<td>41.28%</td>
<td>62.42%</td>
<td>10.62%</td>
<td>50.00%</td>
</tr>
</tbody>
</table>
### Size of Dependence Cycles

<table>
<thead>
<tr>
<th>Function</th>
<th>ORD</th>
<th>VTA</th>
<th>OCFA</th>
<th>OB</th>
<th>OBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress(128)</td>
<td>6,4,4</td>
<td>5,3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>jess(163)</td>
<td>147,4</td>
<td>139</td>
<td>96</td>
<td>94</td>
<td>90</td>
</tr>
<tr>
<td>raytrace(41)</td>
<td>18,4,2</td>
<td>16</td>
<td>8,4</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>db(20)</td>
<td>10</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>javac(184)</td>
<td>169</td>
<td>161</td>
<td>161</td>
<td>142</td>
<td>134</td>
</tr>
<tr>
<td>mpegudio(60)</td>
<td>44</td>
<td>37</td>
<td>6,3,2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mtrt(41)</td>
<td>18,4,2</td>
<td>16</td>
<td>8,4</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>jack(69)</td>
<td>44</td>
<td>7,6,2,2</td>
<td>7,4,2,2</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>jbb(104)</td>
<td>79</td>
<td>49</td>
<td>3,3,2</td>
<td>2,2</td>
<td>2</td>
</tr>
</tbody>
</table>

### Blended Analysis for Performance Diagnosis

- Framework-intensive applications look like *icebergs* to developers
- Problematic activity usually spans multiple *frameworks*
- Long call chains across multiple frameworks often lead to *object churn*
  - Want to identify these
- Current profiling tools focus on object creation rather than object use (e.g. Jinsight, ArcFlow, HPROF)

---

B. Dufour, B.G. Ryder, G. Sevitsky, "Blended Analysis for Performance Understanding of Framework-based Applications", ISSTA'07
Blended Analysis Paradigm

• Need information about a set of program executions
  - Full dynamic analysis is too expensive and too intrusive for production codes
  - Static analysis is too conservative and likely not to scale
• IDEA: Use dynamic analysis to obtain a calling structure of interest to use in a subsequent static analysis
  - Avoids intrusiveness and problems with dynamic loading
  - Dynamic analysis can selectively collect additional useful information (e.g., object creations)

Blended Analysis Paradigm

Dufour et al, ISSTA'07
Definitions

- **Effective lifetime**: period between an object’s creation and its last use.
- **Allocation context**: method invocations on runtime stack during object allocation.
- **Escape**: An object *escapes* method \( f() \), if \( f() \) is in its allocation context and the object can be accessed beyond the lifetime of the invocation of \( f() \).
- **Capture**: An object is *captured* by method \( g() \) if \( g() \) is in its allocation context and the object cannot be accessed beyond the lifetime of the invocation of \( g() \).

Escape Analysis

- Determines escape status of an object at compile time:
  - *Globally escaping*
  - Escaping through arguments and return values (*arg-escaping*)
  - Non-escaping (*captured*)
- Traditional uses in compilation:
  - On-stack allocation
  - Synchronization removal
Blended Escape Analysis

- Based on escape analysis in Choi et. al, TOPLAS’03
- **Connection graphs**
  - Nodes represent objects, fields and references
  - Edges represent points-to relationships
  - Abstract objects are allocation sites
  - Modified to keep a distinct escape state for each object at each node in the calling structure
- Flow-sensitive, context-insensitive, field-sensitive analysis
- Now for an example of static escape analysis...

Example

```java
public X identity(X obj1){
    return obj1;
}

global = obj2;
return obj2;
}

public void f() {
    X inst;
    if (...) inst = identity(new Y())
    else inst = escape(new Z());
}
```

Connection Graph

```
obj1  return

Phantom Object #1

Preserves escape state of actual argument
```
Example

public X identity(X obj1) {
    return obj1;
}

public X escape(X obj2) {
    G.global = obj2;
    return obj2;
}

public void f() {
    X inst;
    if (...) inst = identity(new Y())
    else inst = escape(new Z());
}

Example

public X identity(X obj) {
    return obj;
}

public X escape(X obj) {
    G.global = obj;
    return obj;
}

public void f() {
    X inst;
    if (...) inst = identity(new Y())
    else inst = escape(new Z());
}
Example

```
public X identity(X obj){
    return obj;
}

public X escape(X obj){
    G.global = obj;
    return obj;
}

public void f() {
    X inst;
    if (...) inst = identity(new Y())
    else inst = escape(new Z());
}
```

z is globally escaping
y is captured in f()

Implementation

Dufour et al, ISSTA'07
Benchmarks

- Software
  - Trade 6.0.1
  - Websphere Application Server 6.0.0.1
  - DB2 v8.2.0

- 4 configurations of Trade6
  - Run-time mode: Direct, EJB
  - Access mode: Standard, WebServices

- Tracing a single login transaction after warm-up

Size Comparison for Configurations

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Config</th>
<th>Methods</th>
<th>Invocs</th>
<th>Max Stack</th>
<th>Type</th>
<th>Insts</th>
<th>Abs. Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>getHoldings</td>
<td>Direct-Std</td>
<td>710</td>
<td>4,484</td>
<td>26</td>
<td>30</td>
<td>186</td>
<td>549</td>
</tr>
<tr>
<td></td>
<td>Direct-WS</td>
<td>3,308</td>
<td>127,794</td>
<td>53</td>
<td>166</td>
<td>5,522</td>
<td>2,517</td>
</tr>
<tr>
<td></td>
<td>EJB-Std</td>
<td>1,978</td>
<td>60,936</td>
<td>62</td>
<td>82</td>
<td>1,751</td>
<td>1,834</td>
</tr>
<tr>
<td></td>
<td>EJB-WS</td>
<td>4,479</td>
<td>184,288</td>
<td>72</td>
<td>210</td>
<td>7,088</td>
<td>3,747</td>
</tr>
</tbody>
</table>

getHoldings: Retrieves user's portfolio from a database
Reduced Connection Graphs

- Use additional calling context tree (CCT) data from Jinsight about object instances created, to prune uninteresting object nodes from the connection graphs
- Sum the number of remaining instances in each escape state at method node in the CCT
- Show only CCT nodes that capture object instances, to try to understand use of temporaries
- Used in performance diagnosis

CCT showing only capturing nodes reachable from HoldingDataBean_Ser.serialize in Direct-WS config

Function of CCT: formatting stock records into response portion of SOAP message

262 instances captured in CCT
Reduced Connection Graph for 
DateSerializer.getValueAsString()
Summary

• Presented 3 research projects using program analysis for testing, performance diagnosis and program understanding
  - Showed different cost/benefit tradeoffs for analyses used
  - Demonstrated strength of using static and dynamic analyses together
  - Illustrated the need for empirical investigation with accepted benchmarks
Algorithm Illustration

Methods Callable on CUT

class
Method
Heap Data
Call and return "useful" value
Other calls
Side Effect
write
read

Algorithm Illustration

Methods Callable on CUT

class
Method
Heap Data
Call and return "useful" value
Other calls
Side Effect
write
read
Algorithm Illustration

Methods Callable on CUT

CUT

class
Method
Heap Data
Call and return "useful" value
Other calls
Side Effect
write
read

Breakdown of Abstract Objects by Escape State

Percentage of abstract objects
Always captured
Sometimes captured
Never captured

Percentage of abstract objects
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%

ACACES-5 July 2007 © BG Ryder

ACACES-5 July 2007 © BG Ryder