Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time

McGill University
University of Waterloo
McGill University
Eric Bodden
Patrick Lam
Laurie Hendren
boolean foo(Iterator i, Iterator j) {
  while (i.hasNext() && i.hasNext()) {
    if (i.next() != j.next())
      return false;
  }
  return true;
}

boolean foo(Iterator i, Iterator j) {
  while (i.hasNext() && i.hasNext()) {
    if (i.next() != j.next())
      return false;
  }
  return true;
}
Problem 1:
Potentially large runtime overhead
Problem 2:
Dynamic, with no static guarantees
boolean foo(Iterator i, Iterator j){
    while(i.hasNext() && j.hasNext()){
        if(i.next() != j.next())
            return false;
    }
    return true;
}
Problem 3: Existing sound static approaches have many false positives
boolean foo(Iterator i, Iterator j) {
    while (i.hasNext() && j.hasNext()) {
        if (i.next() != j.next())
            return false;
    }
    return true;
}
Most simple example: HasNext

Don't call `next()` twice on an `Iterator i` without calling `hasNext()` in between!
Tracematch HasNext

tracematch(Iterator i) {

}

Allan et al., OOPSLA 05
Tracematch HasNext

tracematch(Iterator i) {
    sym hasNext after returning:
        call(* Iterator.hasNext()) && target(i);
    sym next after returning:
        call(* Iterator.next()) && target(i);
}

Allan et al., OOPSLA 05
Tracematch HasNext

tracematch(Iterator i) {
    sym hasNext after returning:
    call(* Iterator.hasNext()) && target(i);
    sym next after returning:
    call(* Iterator.next()) && target(i);

    next next

}

Allan et al., OOPSLA 05
```java
tracematch (Iterator i) {
    sym hasNext after returning:
    call(* Iterator.hasNext()) && target(i);
    sym next after returning:
    call(* Iterator.next()) && target(i);

    next next {
        System.out.println("Called ‘next’ twice on" +i+"!");
    }
}
```

Allan et al., OOPSLA 05
next, hasNext

true \quad i = f(a(s1)) \quad i = f(a(s1))

```java
{
    System.out.println(
        "Called 'next' twice on" + i + "!");
}
```
Novel static program analyses
Trade-off: Speed vs. Precision

```c
void foo(Iterator i) {
}
```
For every program variable i:

Can program variable j point to the same object as i?

```
void foo(Iterator i) {
    // Code goes here
}
```
Problem 1: Missing info at method entry

```java
void foo(Iterator i) {
    i.next();
}
```
Key observation!

next, hasNext

Possible targets of “hasNext” is a state-determining symbol!
Key observation!

Benchmarks: 68% of symbols are state-determining
Problem 1: Missing info at method entry

```java
void foo(Iterator i) {
    if (i.hasNext()) {
        i.next();
    }
}
```
Problem 2: Aliasing

```java
void foo(Iterator i) {
    if (i.hasNext()) {
        Iterator i2 = i;
        i2.next();
    }
}
```
Object representatives

Fl: Must (Dummy) Must-not-alias (points-to)

Don't know

FS: Must-alias Must-not-alias

o1 o2

Current method

Must Must-not

o3

Other method

Precision where we need it

Speed where we need it
Problem 3: Outgoing method calls?

```java
void foo(Iterator i) {
    if (i.hasNext()) {
        bar(i);
        i.next();
    }
}
```

**Summary Information**

*bar(i) may only call hasNext() on i!*
Problem 4: Continuation

void foo(Item i) { 
  if (i.hasNext()) {
    i.next();
  }
}

Can we remove the instrumentation here? NO!

No missed violations at runtime!

Summary Information

Rest of program may call next() on i!
Let the fun begin…

binding multiple objects!

For every Collection \( c \) and Iterator \( i \):

Don't modify \( c \) while \( i \) is used on \( c \).
void whiz(Collection c1) {

  Iterator il = c1.iterator();

  il.next();
}

Let the fun begin... binding multiple objects!
Solution: "Uniqueness Check"

Prove that:

\[ i = o(i1) \rightarrow c = o(c1) \]

Summary Information

Requires clever combination of pointer analyses (using object representatives).
## Benchmarks - Tracematches

<table>
<thead>
<tr>
<th>ASynclIteration</th>
<th>HasNextElem</th>
</tr>
</thead>
<tbody>
<tr>
<td>FailSafeEnum</td>
<td>LeakingSync</td>
</tr>
<tr>
<td>FailSafeIterator</td>
<td>Reader</td>
</tr>
<tr>
<td>HashMap</td>
<td>Writer</td>
</tr>
<tr>
<td>HasNext</td>
<td></td>
</tr>
</tbody>
</table>
## Benchmark programs

**DaCapo:**

<table>
<thead>
<tr>
<th>antlr</th>
<th>hsqldb</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat</td>
<td>jython</td>
</tr>
<tr>
<td>chart</td>
<td>lucene</td>
</tr>
<tr>
<td>eclipse</td>
<td>pmd</td>
</tr>
<tr>
<td>fop</td>
<td>xalan</td>
</tr>
</tbody>
</table>

... and SciMark (with 4 extra tracematches)
Results – Elimination of potential failure points

103 program/tracematch combinations
static guarantees in 84 cases
in 14 cases: less than 10 potential failure points
Classification of potential failure points

- Likely
  - \texttt{foo(\ldots)} line 23

- Unlikely
  - \texttt{bar(\ldots)} line 42
  - \texttt{bar(\ldots)} line 44
Analysis annotates potential failure points

HasNext:
next - <InductionVarAnalyzer.isMu(..)> @ line 217

next - <InductionVarAnalyzer.isMu(..)> @ line 218

HasNext:
next - <CodeGen.removeEmptyBl(..)> @ line 587
hasNext - <CodeGen.removeEmptyBl(..)> @ line 586
Analysis annotates potential failure points

HasNext: features []
next - <InductionVarAnalyzer.isMu(..)>
  @ line 217
next - <InductionVarAnalyzer.isMu(..)>
  @ line 218

HasNext: features [CALL]
next    - <CodeGenerator.removeEmptyBl(..)>
  @ line 587
hasNext - <CodeGenerator.removeEmptyBl(..)>
  @ line 586
Features: Reasons for imprecision

- CALL
- ABORTED
- NO_CONTEXT
- DELEGATE
- CONTINUATION
- DYNAMIC_LOADING
- OVERLAPS
Manually annotated actual failure points

HasNext: features [], ACTUAL
next - <InductionVarAnalyzer.isMu(..)>
  @ line 217
next - <InductionVarAnalyzer.isMu(..)>
  @ line 218

HasNext: features [CALL]
next - <CodeGenerator.removeEmptyBl(..)>
  @ line 587
hasNext - <CodeGenerator.removeEmptyBl(..)>
  @ line 586
Weka machine learning kit

CALL = 0
| ABORTED = 0
| | DELEGATE = 0
| | | NOCONTEXT = 0: TRUE_POSITIVE (11.0/1.0)
| | | NOCONTEXT = 1: FALSE_POSITIVE (4.0/1.0)
| | DELEGATE = 1: FALSE_POSITIVE (10.0)
| ABORTED = 1: FALSE_POSITIVE (30.0)
CALL = 1: FALSE_POSITIVE (406.0/1.0)
Results – Filtering

Found 5 programs with bugs or questionable code.

- Correct
- Un-filtered false positive
- Filtered actual violation
Related work: Typestate

Static and hybrid verification of typestate props.

- Typestate (Strom & Yemini, TSE Vol 12 No. 1, 86)
- Fugue for .NET (DeLine & Fähndrich, ECOOP 04)
- Typest. & Aliasing (Bierhoff & Aldrich, OOPSLA 07)
- Hybrid static/dynamic (Dwyer & Purandare, ASE 07)
Related work: Tracematch-like

Flow-sensitive analysis of Tracematches

- Naeem and Lhoták, OOPSLA 08

Other state-based runtime-verification tools for Java

- JavaMOP (Chen & Roșu, OOPSLA 08)
- PQL (Martin, Livshits & Lam, OOPSLA 05)
- PTQL (Goldsmith, O’Callahan & Aiken, OOPSLA 05)
Related work: Static checkers

Static checkers
- FindBugs (Hovemeyer & Pugh, OOPSLA 04)
- PMD (http://pmd.sf.net/)

Pre and postconditions, invariants
- ESC/Java (Flanagan et al., PLDI 02)
- Java Modeling Language (JML)

Specialized interprocedural analyses
- Jlint (http://archo.com/jlint/)

Comparison: Rutar et al., ISSRE 04
Related work: Invariant mining and checking

Dynamic invariant inference and checking
- Daikon (Ernst et al., TSE Vol 27. No 2, 01)
- DIDUCE (Hangal & Lam, ICSE 02)
- JADET (Wasylkowski et al., FSE 07)
- Spec. Mining (Ammons et al., POPL 02)

Static rule mining and checking
- PR-Miner (Li & Zhou, FSE 05)
- Houdini (Flanagan & Leino, FME 01)
Special thanks to...

Co-workers
- Ondřej Lhoták
- Nomair Naeem

Maintainers of Tracematch implementation
- Pavel Avgustinov
- Julian Tibble
private final void FillBuff() {
    ...
    try {
        if ((i = inputStream.read(...)) == -1) {
            inputStream.close();
            throw new java.io.IOException();
        }
        else
            maxNextCharInd += i;
        return;
    }
    ...
}
```java
static String getLine(BufferedReader reader, int line) {
    if (reader == null)
        return "";
    try {
        String text = null;
        for (int i = 0; i < line; i++) {
            text = reader.readLine();
        }
        return text;
    } catch (IOException ioe) {
        return null;
    }
}
```
public Block isMu(...) {
...
    final Iterator iter = cfg.preds(phi.block()).iterator();
    final Block pred1 = (Block) iter.next();
    final Block pred2 = (Block) iter.next();
private List markUsages(IDataFlowNode inode) {
    ...
    for (Iterator k = ((List)entry.getValue()).iterator(); k.hasNext();) {
        addAccess(k, inode);
    }
    ...
}

private void addAccess(Iterator k, IDataFlowNode inode) {
    NameOccurrence occurrence = (NameOccurrence) k.next();
    ...
}
private List markUsages(IDataFlowNode inode) {
    ...
    for (NameOccurrence occurrence: entry.getValue()) {
        addAccess(occurrence, inode);
    }
    ...
}

private void addAccess(NameOccurrence occurrence, IDataFlowNode inode) {
    ...
}
while (c == null && enumMap.hasMoreElements()) {
    ...
    if (!enumC.hasMoreElements())
        c = null;
}
// At this point, c == null if there are no more elements,
// and otherwise is the first collection with a free element
// (with enumC set up to return that element).
if (c == null) {
    // no more elements, so return null;
    return (null);
} else {
    Perm answer = (Perm) enumC.nextElement();
    ...
}
public Iterator iterator() {
    return new Iterator() {
        Iterator i = list.iterator();
        public void remove() {
            throw new UnsupportedOperationException();
        }
        public boolean hasNext() {
            return i.hasNext();
        }
        public Object next() {
            return i.next();
        }
    };
}

Results – Static analysis time

Average total: 6 minutes
Max total: 20 minutes
Delegating calls

```java
public Object next()
{
    DELEGATE
    inner.next()

    // Diagram arrows pointing to public Object next() outside
}
```
Reasons for imprecision

```java
boolean foo(Iterator i, Iterator j) {
    bar(i);
    void bar(..)
}
```
reasons for imprecision

```java
boolean foz(Set c1, Set c2) {
    Iterator i1 = c1.iterator();
    public Iterator iterator() {
        return new HashIterator();
    }
    Iterator i2 = c2.iterator();
}
```

NO_CONTEXT
boolean baz(Iterator i, Iterator j) {

}

ABORTED
Using alias queries to reduce false-positive rate

Assume we know **r1 and r2 must-alias**, r1 occurs in some constraint bound to x and we see an event that binds x to r2.

<table>
<thead>
<tr>
<th>x = r2</th>
<th>x = r1</th>
<th>x ≠ r1</th>
<th>x ≠ r2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x = r1 ≡ x = r2</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td></td>
<td>false</td>
<td></td>
<td>x ≠ r1 ≡ x ≠ r2</td>
</tr>
</tbody>
</table>
Using alias queries to reduce false-positive rate

Assume we know \( r1 \) and \( r2 \) must-not-alias, \( r1 \) occurs in some constraint bound to \( x \) and we see an event that binds \( x \) to \( r2 \).

<table>
<thead>
<tr>
<th>( x = r2 )</th>
<th>( x = r1 )</th>
<th>( x \neq r1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = r2 )</td>
<td>\textit{false}</td>
<td>( x = r2 )</td>
</tr>
<tr>
<td>( x \neq r2 )</td>
<td>( x = r1 )</td>
<td>( x \neq r1 \land x \neq r2 )</td>
</tr>
</tbody>
</table>