Using FindBugs™ for Research

David Hovemeyer and William Pugh

June 10, 2007
Introduction

Start with a bug

Implementing detectors and analyses

Tracking warnings

Tools
Programmers are smart

We have good techniques (e.g., unit testing, pair programming, code inspections) for finding bugs early

So, most bugs remaining in production code must be subtle, and require sophisticated techniques to find

Right?
if (listeners == null)
    listeners.remove(listener);
public String foundType() {
    return this.foundType();
}
try { ... } 
catch (IOException e) {
    new SAXException("Server side Exception:" + e);
}
Finding bugs is easy

- Finding bugs is easy *once you know what to look for*
- **FindBugs\textsuperscript{TM}:** [http://findbugs.sourceforge.net](http://findbugs.sourceforge.net)
  - Detectors for wide range of common Java bug patterns
  - Open source
  - Easy to use, large existing user base
  - Extensible, modular
Why FindBugs for static analysis research?

Working infrastructure for:

- User interface
- Bytecode-level static analysis
- XML-based format for saved warnings
  - Handling multiple versions of software
  - Tracking warnings over time

You can focus on developing new analyses, empirical studies, etc.
Large user base: find out if your idea works in the “real world”
Research Ideas...

- Implement new bug detectors
  - API specific (e.g., structs, Swing, Hibernate)
  - General purpose (e.g., deadlock)
- Annotations for defect detection
- Explore how developers interact with static analysis
More Research Ideas...

- Do historical analysis
  - When do warnings get introduced, removed
  - Correlation between unit tests and field failures and defect warnings
  - Correlation code coverage, churn, metrics and warnings
  - Correlation between style and defect warnings
- Improve presentation/UI for static analysis
- When does bad code cause the software to behave badly?
This tutorial assumes

- Knowledge of Java
- Basic understanding of Java classfiles and Java bytecode
- Basic understanding of compiler and static analysis techniques

For more information:

- Any compilers textbook
- VMSpec, 2nd ed.: http://java.sun.com/docs/books/jvms/
Acknowledgements

FindBugs is sponsored by:

Fortify

Surelogics

Additional support provided by: Google, Sun Microsystems
And all of the contributors to the open source effort
Demo
Introduction

Start with a bug

Implementing detectors and analyses

Tracking warnings

Tools
Implementation techniques

- There are many ways to implement a FindBugs detector
- And no “right way”
Basic approach to writing a detector

1. Start with a bug (important!)
2. Write the simplest possible detector that might find similar bugs
3. Evaluate: does it find enough interesting bugs without too many false positives?
4. Refine: improve analysis and FP suppression heuristics
5. Repeat steps 3 and 4 until you get something acceptable or you give up on the idea
Don’t use String literals for the synchronized blocks! http://www.javalobby.org/java/forums/t96352.html

Code that is using String literals to synchronize on is dangerous.

```java
static private final String LOCK = "LOCK";
void someMethod() {
    synchronized(LOCK) {
        ...
    }
}
```

...something like this really happened between our code and code in the Jetty library. Both sections used the same string literal above to synchronize critical code sections. The two code segments created a dead-lock with very puzzling stack traces (The Jetty-Bug has been reported already, btw, Jetty-352).
Write a detector to look for synchronization on constant Strings

- Add test case
- Bytecode:
  
  LDC "LOCK"
  DUP
  ASTORE 1
  MONITORENTER
- Let’s use opcode stack.
  - Could also look for bytecode sequence.
public void sawOpcode(int seen) {
    if (seen == MONITORENTER) {
        OpcodeStack.Item top = stack.getStackItem(0);
        if (top.getSignature().equals("Ljava/lang/String;"))
            && top.getConstant() instanceof String)
            bugReporter.reportBug(new BugInstance(this,
                "DL_SYNCHRONIZATION_ON_SHARED_CONSTANT",
                NORMAL_PRIORITY)
                .addClassAndMethod(this)
                .addString((String) constant)
                .addSourceLine(this));
    }
}
Results

- Found issue fixed in Jetty-352.
- Jetty-352 didn’t fix all occurrences in Jetty (Jetty-362).
- Also found occurrences in Eclipse, glassfish, Sun’s JDK, netbeans, nutch, oc4j, weblogic, websphere.
- Not bad for 20 minutes work.
Why simple techniques work

- We aren’t trying to prove anything about the code (other people do that).
- Simple mistakes generally result in mistakes that are easy to find.
- javac does minimal optimization/ transformation
- Simple analysis produces results that are easy to triage.
All FindBugs detectors work by analyzing bytecode. Supported frameworks:

- BCEL (http://jakarta.apache.org/bcel/); DOM-like API
- ASM (http://asm.objectweb.org/); SAX-like API

Currently, much of the supporting FindBugs infrastructure is based on BCEL. Support for ASM-based analyses and detectors is experimental.
Types of detectors

Most FindBugs detectors use one of the following implementation techniques:

- Inspecting class/method/field structure
- Micropatterns: simple bytecode patterns
- Stack-based patterns
- Dataflow analysis
- Interprocedural analysis

Each implementation technique is supported by ready-made base classes and support infrastructure.
Some detectors do not require code analysis. Examples:

- Find classes that override `equals()` but not `hashCode()`
- Find method naming problems (e.g., `hashcode()` instead of `hashCode()`)

Typical implementation strategy: detector class extends `PreorderVisitor`

- Override `visit(JavaClass)`, `visit(Method)`, `visit(Field)`, etc.
- Report problems
Micropatterns: simple bytecode patterns

Look for simple instruction sequences:

- E.g., Find unconditional wait

**Source code**

```java
synchronized (lock) {
    lock.wait();
    ...
}
```

**Bytecode in class file**

```java
ALOAD 0
GETFIELD A.lock
DUP
ASTORE 1
MONITORENTER
ALOAD 0
GETFIELD A.lock
INVOKEVIRTUAL Object.wait()V
```
Introduction
Start with a bug
Implementing detectors and analyses
Tracking warnings
Tools
Conclusions

Basic approach
Strategies

David Hovemeyer and William Pugh

Using FindBugs™ for Research
Stack-based patterns

Micropatterns where the values on the operand stack are significant. Examples:

- As seen earlier: look for `monitorenter` on constant String value

Typical implementation strategy:

- Detector class extends `OpcodeStackVisitor`, overrides `visit(Method)` and `sawOpcode()`
- `sawOpcode()` uses the `OpcodeStack` object stored in `stack` field to inquire about values on operand stack
- Warn when suspicious instruction sequence/stack values seen
Dataflow analysis

Use intraprocedural dataflow analysis to infer (probable) facts within methods. Examples:

- Find dereferences of null values
- Find field accesses not consistently protected by a lock

Typical implementation strategy:

- Detector class extends CFGDetector
- For visited application classes/methods, request dataflow analysis results from AnalysisCache
- Inspect code for suspicious situations (e.g., dereference of possibly-null value)
Interprocedural

Summarize method behavior, and use that summary at each call site. Examples:

- Method parameters that are unconditionally dereferenced.
- Return values that are always nonnull.
- Methods that always throw an exception.

We don’t (currently) do any context sensitive interprocedural analysis.
Interprocedural issues

- Resolving virtual and interface method invocations.
- Order in which to evaluate methods
  - If you can separate computation and use into separate analysis passes, do so.
  - Right now, FindBugs sorts classes, and within each class, sorts methods, according to call graph.
  - Works fairly well.
Introduction

Start with a bug

Implementing detectors and analyses

Tracking warnings

Tools
Important concepts. Each is a class or interface:

- Project
- IAnalysisCache (interface)
- ClassDescriptor / MethodDescriptor
- Detector2 (interface)
- ExecutionPlan/AnalysisPass
- BugInstance/BugAnnotation
- BugReporter (interface)
- BugCollection

These classes (and their subclasses) form the core FindBugs object model.
Project: a Java software artifact to be analyzed

- Application classes: scanned for bugs
- Auxiliary classes: classes referenced by application classes, but not scanned for bugs
  - Needed to accurately model exceptions, inheritance hierarchy, etc.
- Source directories: allow mapping of reported warnings back to Java source code
IAnalysisCache (interface)

IAnalysisCache: global cache of *analysis objects* and *databases*

- An *analysis object* stores facts (or probable facts) about a class or method
  - E.g., an in-memory object representing the class’s bytecode
  - E.g., the results of a null-pointer dataflow analysis on a method

- A *database* stores facts about the entire program
  - E.g., which methods unconditionally dereference parameters?
All analysis objects are referred to using a ClassDescriptor or MethodDescriptor.

- **ClassDescriptor**: identifies a class
- **MethodDescriptor**: identifies a method
Detector2: implemented by classes which scan application classes for bugs

- We refer to such classes as “detectors”
- `visitClass` method invoked on each application class
- General idea:
  - Request one or more *analysis objects* from the AnalysisCache for the class and its methods
  - Inspect analysis objects
  - Report warnings for suspicious situations in code
ExecutionPlan/AnalysisPass

ExecutionPlan: list of AnalysisPasses specifying how to apply detectors to Project
AnalysisPass: one pass executing Detectors over the application classes

- Before starting analysis of the Project, FindBugs divides the Detectors into AnalysisPasses
- Plugin can register ordering constraints
  - E.g., Detector A must run in an earlier pass than Detector B
  - E.g., Detector C must run before Detector D in same pass

David Hovemeyer and William Pugh
Using FindBugs™ for Research
BugInstance/BugAnnotation

BugInstance: a reported static analysis warning
BugAnnotation: information about a warning

- BugInstance consists of type, priority, and BugAnnotations
  - Type: what kind of bug is suspected
  - Priority: how severe?
- BugAnnotations describe
  - Where the issue occurs (class, method, bytecode, source file, line number)
  - Other information

David Hovemeyer and William Pugh
Using FindBugs™ for Research
BugReporter: object through which BugInstances are reported

- When a Detector is instantiated, its constructor gets a reference to a BugReporter
- The Detector remembers its BugReporter and uses it to emit warnings
- Usually, the BugReporter will save reported BugInstances in a BugCollection
BugCollection: a collection of BugInstances

- Basic use of FindBugs:

  Project → findbugs → BugCollection

- A BugCollection can contain BugInstances from many versions of same software artifact:

  BugCollection\textsubscript{t} → compute BugHistory → BugCollection\textsubscript{0..t} → BugCollection\textsubscript{0..t-1}
When FindBugs runs

What happens when FindBugs runs?

1. Read in project
2. Find all application classes in the project
3. Load plugins containing detectors
4. Create an execution plan
5. Run the FindBugs algorithm to apply detectors to application classes
FindBugs algorithm (pseudo-code)

```
for each analysis pass in the execution plan do
  for each application class do
    for each detector in the analysis pass do
      apply the detector to the class
    end for
  end for
end for
```
FindBugs algorithm details

- Detectors in first AnalysisPass applied to not only application classes but also *referenced* classes
  - E.g., collect annotations in referenced classes/methods so uses can be checked
- Databases can be used to convey information between analysis passes
  - Detector in earlier pass creates and populates database, detector in later pass retrieves information
In this section we will write a FindBugs plugin with two FindBugs detectors (micropattern and dataflow-based).

Goals:
- FindBugs plugin structure
- Intro to FindBugs internals
- Standard detector implementation techniques

You can follow along!
- Handout zip drive / CD contains recent FindBugs source dist plus two versions of demo plugin source
Demo plugin

- demopluginsrc_start.zip: empty “starting point” plugin
- demopluginsrc_finished.zip: the completed demo plugin
- Demo plugin must be checked out in sibling directory of main FindBugs source dist
  - Recommended: use Eclipse
- Demo plugin code is in public domain
  - You are encouraged to use it as the basis for your own plugins
  - Just change package names and other identifying information so that it does not conflict with edu.umd.cs.findbugs.* and net.findbugs.*
Steps in writing a detector

- Create plugin (or add to existing plugin)
  - Descriptor file
  - Messages file
- Add detector class to plugin
- Add any required analysis classes to plugin
- Add detector and bug patterns to plugin descriptor
- Add messages for plugin, bug pattern, and bug codes to messages file
- Install plugin in a FindBugs installation
Plugin structure
What is a plugin?

- A plugin is a jar file containing detector classes and analysis classes
  - Plus some meta-information
- To install the plugin, just copy the jar file into the plugin subdirectory of a FindBugs installation
- Plugins allow FindBugs functionality to be extended without any code changes
- Plugins can use any of the built-in FindBugs analyses
Plugin overview

Components of a plugin

- Detector and analysis classes
- `findbugs.xml`: plugin descriptor
  - Detector ordering constraints
  - Detector classes
  - Analysis engine registrar
  - Bug patterns
- `messages.xml`: human-readable messages
  - Required: default messages file (English)
  - Optional: translations for other languages, e.g. `messages_ja.xml`
findbugs.xml skeleton

Blue = things you will edit

```xml
<?xml version="1.0" encoding="UTF-8"?>
<FindbugsPlugin pluginid="net.findbugs.demoplugin"
    defaultenabled="true"
    provider="FindBugs project"
    website="http://findbugs.sourceforge.net">

Detector ordering constraints

Detectors

Analysis engine registrar (optional)

Bug patterns
</FindbugsPlugin>
```
Blue = things you will edit

<?xml version="1.0" encoding="UTF-8"?>
<MessageCollection>
  <Plugin info>
  </Plugin info>
  <Detector info>
  </Detector info>
  <Bug pattern info and messages>
  </Bug pattern info and messages>
  <Bug code info and messages>
  </Bug code info and messages>
</MessageCollection>
Micropattern detector
Micropattern detector example

- Find calls to System.gc()
  - Bad idea in general
  - Really bad idea in finalizers
  - We’ll allow them in main methods

- FindBugs already detects this bug pattern in the DumbMethods detector

- Analysis technique: detector just needs to find invokestatic instructions that call this method
Add detector class

![New Java Class dialog box](image)

- **Source folder:** `demoplugin/src`
- **Package:** `net.findbugs.demo.detect`
- **Name:** `FindSystemGCInvocations`
- **Modifiers:**
  - Public
- **Superclass:** `edu.umd.cs.findbugs.BytecodeScanningDetector`
- **Interfaces:**
Detector boilerplate

- All detectors must have a constructor that takes a BugReporter as a parameter
- Save it in a field for later use

```java
private BugReporter bugReporter;

public FindSystemGCInvocations(BugReporter bugReporter) {
    this.bugReporter = bugReporter;
}
```
Analysis technique

- Override `sawOpcode` (instruction-level visitor callback)
- Look for `invokestatic` instructions
- Warn when
  1. Class = `java.lang.System`
  2. Method = `gc`
  3. Signature = `()V`
sawOpcode() implementation

```java
@Override
public void sawOpcode(int seen) {
    if (seen != Constants.INVOKESTATIC) { return; }

    String calledClassName = getClassConstantOperand();
    String calledMethodName = getNameConstantOperand();
    String calledMethodSig = getSigConstantOperand();

    if (calledClassName.equals("java/lang/System")
        && calledMethodName.equals("gc")
        && calledMethodSig.equals("()V") ) {
        emitWarning();
    }
}

private void emitWarning() {
    System.out.println("Warn about " + getMethodName()); // TODO
}
```
Add detector to findbugs.xml

For FindBugs to see the detector, it must be registered in the plugin descriptor, findbugs.xml

```xml
<Detector
class="net.findbugs.demo.detect.FindSystemGCInvocations"
reports=""
hidden="false" />
```

Now, run the install target in build.xml to build the plugin and install in FindBugs plugin directory
Verify that plugin can be loaded

- In shell, add bin subdirectory to PATH
- Run `findbugs -textui -showPlugins`
- Should see something like this:

  Available plugins:
  - `edu.umd.cs.findbugs.plugins.core` (default: enabled)
    - Description: Default FindBugs plugin
    - Provider: FindBugs project
    - Website: http://findbugs.sourceforge.net
  - `net.findbugs.demoplugin` (default: enabled)
    - Description: Demonstration FindBugs plugin
    - Provider: FindBugs project
    - Website: http://findbugs.sourceforge.net
public class TestSGC {
    protected void finalize() throws Throwable {
        System.gc(); // very bad
    }

    public void test() {
        System.gc(); // dumb
    }

    public static void main(String[] args) {
        System.gc(); // maybe this is ok; don’t report a warning
    }
}
First test of detector

- In demoplugin directory, cd to bin/net/findbugs/demo/testcode subdirectory
- Run findbugs -textui -visitors \FindSystemGCInvocations *.class
- Should see debug output from detector:
  - Warn about finalize
  - Warn about test
  - Warn about main
To emit warnings properly, we need to define a bug pattern. “Bug pattern” = specific kind of bug

- Registered in findbugs.xml
- Associated messages in messages.xml
- At runtime, occurrences (warnings) reported via the BugReporter
Add bug pattern to findbugs.xml

- Edit reports attribute of Detector element
  
  ```xml
  <Detector
class="net.findbugs.demo.detect.FindSystemGCInvocations"
reports="SGC_SYSTEM_GC"
hidden="false" />
  ```

- Add BugPattern element
  
  ```xml
  <BugPattern abbrev="SGC" type="SGC_SYSTEM_GC"
category="BAD_PRACTICE"/>
  ```
Add bug pattern and bug code to messages.xml

- Add BugPattern element
  
  ```xml
  <BugPattern type="SGC_SYSTEM_GC">
    <ShortDescription>Call to System.gc()</ShortDescription>
    <LongDescription>Call to System.gc() in method \{1\}</LongDescription>
    <Details>
      <![CDATA[<p>Call to System.gc(). Not a great idea.</p>]]>
    </Details>
  </BugPattern>
  ```

- Add BugCode element
  
  “Bug code” = group of related bug patterns
  
  ```xml
  <BugCode abbrev="SGC">Call to System.gc()</BugCode>
  ```
BugPattern element structure

Structure of a BugPattern element in messages file

- **ShortDescription**: one line summary of warning
- **LongDescription**: one line summary of warning with filled-in BugAnnotations
  - Placeholder \{n\} refers to nth BugAnnotation
  - Can refer to class, method, field
  - By convention: 0 is class, 1 is method
  - Example: Call to System.gc() in \{1\}
- **Details**: raw XHTML describing the bug pattern
  - Tags must be lower case and properly balanced (don’t omit end tags)
Emitting a warning

- Change `emitWarning` to use the BugReporter to emit a warning:
  - Create BugInstance object, decorate with BugAnnotations
  - Call reportBug method on BugReporter

```java
private void emitWarning() {
    String type = "SGC_SYSTEM_GC";
    int priority = Priorities.NORMAL_PRIORITY;
    BugInstance warning = new BugInstance(this, type, priority)
        .addClassAndMethod(this)
        .addSourceLine(this);
    bugReporter.reportBug(warning);
}
```
Second test of detector

- Run `findbugs -textui -visitors \ FindSystemGCInvocations *.class`

- Output:

  M B SGC: Call to System.gc() in method net.findbugs.demo.testcode. TestSGC.finalize() At TestSGC.java:[line 12]
  M B SGC: Call to System.gc() in method net.findbugs.demo.testcode. TestSGC.test() At TestSGC.java:[line 16]
  M B SGC: Call to System.gc() in method net.findbugs.demo.testcode. TestSGC.main(String[]) At TestSGC.java:[line 20]

Warnings are now reported properly; can display in GUI, save in XML, etc.
However, we want to report occurrences in finalizers at a high priority and suppress reporting of occurrences in main methods.
Keeping track of analyzed method

- Add fields to keep track of whether we are in a main or finalize method:
  ```java
  private boolean inMain;
  private boolean inFinalizer;
  ```

- Override visit(Method):
  ```java
  @Override
  public void visit(Method obj) {
      inMain = obj.getName().equals("main");
      inFinalizer = obj.getName().equals("finalize");
  }
  ```

  `visit(Method)` is called before `sawOpcode`.
private void emitWarning() {
    if (inMain) {
        return;
    }
    String type = inFinalizer ?
        "SGC_SYSTEM_GC_IN_FINALIZER" : "SGC_SYSTEM_GC";
    int priority = inFinalizer ?
        Priorities.HIGH_PRIORITY : Priorities.NORMAL_PRIORITY;
    BugInstance warning = new BugInstance(this, type, priority)
        .addClassAndMethod(this)
        .addSourceLine(this);
    bugReporter.reportBug(warning);
}
Update findbugs.xml

- Modify Detector element:
  
  ```xml
  <Detector class="net.findbugs.demo.detect.FindSystemGCInvocations"
    reports="SGC_SYSTEM_GC,SGC_SYSTEM_GC_IN_FINALIZER"
    hidden="false" />
  ```

- Add new BugPattern element:
  
  ```xml
  <BugPattern abbrev="SGC" type="SGC_SYSTEM_GC_IN_FINALIZER"
    category="CORRECTNESS"/>
  ```
Add new bug pattern to messages.xml

<BugPattern type="SGC_SYSTEM_GC_IN_FINALIZER">
    <ShortDescription>Call to System.gc() in finalizer</ShortDescription>
    <LongDescription>Call to System.gc() in method {1}</LongDescription>
    <Details>
        <![CDATA[
            <p>Call to System.gc() in a finalizer. Bad, bad, bad!</p>
        ]]>  
    </Details>
</BugPattern>
Final test of detector

- Run `findbugs -textui -visitors \n  FindSystemGCInvocations *.class`

- Output:

  H C SGC: Call to System.gc() in method net.findbugs.demo.testcode.TestSGC.
  finalize() At TestSGC.java:[line 12]
  M B SGC: Call to System.gc() in method net.findbugs.demo.testcode.TestSGC.
  test() At TestSGC.java:[line 16]
Dataflow-based detector
Inconsistent synchronization detector

- Inconsistent synchronization: look for fields sometimes locked and sometimes unlocked when accessed
- Implemented as FindInconsistentSync2 detector in FindBugs
- Analysis technique: dataflow analysis to count number of locks acquired by monitorenter instructions
- We will ignore many details:
  - Accesses in constructors and other lifecycle methods not likely to be accessible by multiple threads
  - Accesses in a method may be protected by lock acquired in caller
  - Etc.
- Goal: understand dataflow analysis framework in FindBugs
Dataflow analysis concepts

- Symbolically execute method
  - Using CFG, BasicBlocks, InstructionHandles as representation of method

- Estimate facts that are true at each location in method

- Facts (dataflow values) form a lattice
  - Meet operator ($\Diamond$) models merging of control paths
  - Top value ($\top$): $\text{Value} \diamond \top = \text{Value}$ for any Value
  - Bottom value ($\bot$): $\text{Value} \diamond \bot = \bot$ for any Value

- Block transfer function models effect of executing a basic block
  - Edge transfer function models effect of taking a control edge
Control flow graphs

FindBugs BCEL-based intermediate representation:

- CFG: method control flow graph
  - Composed of BasicBlocks and Edges
  - Entry block: dominates every block
  - Exit block: postdominates every block
- BasicBlock composed of list of InstructionHandles
- Each InstructionHandle has reference to Instruction
- Edge connects a source block and target block
  - Edges have edge types: e.g., goto, ifcmp, fall through, handled exception, unhandled exception, etc.
Control flow graph pruning

- FindBugs attempts to “prune” CFGs by removing unlikely/impossible control edges
- This may create dead code!
  - You will notice this when your dataflow analysis completes, but some basic blocks do not have valid facts (all facts are Top)
Exceptions

- Any instruction that can throw an exception is preceded by an exception thrower block (ETB)
  - Exception edges connect an ETB to
    - An exception handler block (handled exceptions)
    - The CFG exit block (unhandled exceptions)
- ETB blocks can be distinguished by calling the isExceptionThrower() method
- An Edge’s edge type can be accessed by calling the getType() method, returns a constant defined in EdgeTypes interface
  - Can also call isExceptionEdge()
jsr subroutines

- The `jsr` bytecode instruction jumps to a subroutine within a method
  - Used by some versions of javac when generating code for a finally block
- FindBugs inlines all `jsr` subroutines into the method control flow graph
  - Greatly simplifies implementation of dataflow analysis
  - Means that an InstructionHandle may be part of multiple BasicBlocks
- Location = BasicBlock + InstructionHandle
  - Marks location of a particular instruction in a particular basic block
Dataflow analysis in FindBugs

- Any class may be used to represent dataflow values
  - Dataflow values must be mutable
- DataflowAnalysis subclass implements dataflow operations:
  - Creating facts
  - Copying facts
  - Computing meet of two facts
  - Block transfer function
  - Etc.
- Dataflow subclass implements dataflow algorithm, provides accessor methods for analysis results
Several base classes are provided for default implementations of important methods:
Important dataflow analysis methods

- createFact: create a new dataflow fact object
- copy: make destination fact object identical to the source fact object
- initEntryFact: get the fact true at the beginning of the CFG entry block
- makeFactTop: set a fact object so that it is equal to the Top fact
- isTop: return whether or not given fact is equal to the Top fact
- same: return whether or not two fact objects are equal in value
Important dataflow analysis methods (continued)

- **transfer**: transfer function to model effect of instructions in a BasicBlock
- **transferInstruction**: for subclasses of AbstractDataflowAnalysis, model the effect of executing one instruction in a BasicBlock
- **edgeTransfer**: transfer function to model effect of a control edge
- **meetInto**: meet function
LockCountAnalysis design

- Java monitor locks are recursive: can be acquired multiple times
- Dataflow fact is an integer count
- When `monitorenter` is executed, increment count
- When `monitorexit` is executed, decrement count
- `javac` ensures that a lock is released no matter how control leaves a `synchronized` block
  - So, we should see proper balancing of `monitorenter / monitorexit`
LockCount lattice
LockCount class

public class LockCount {
    public static final int TOP = -1;
    public static final int BOTTOM = -2;

    private int count;

    public LockCount(int count) { this.count = count; }

    public boolean isTop() { return count == TOP; }
    public boolean isBottom() { return count == BOTTOM; }

    public void setCount(int count) { this.count = count; }

    public int getCount() { return count; }

    @Override
    public String toString() { return String.valueOf(count); }
}

LockCountAnalysis implementation
LockCountAnalysis implementation (continued)

- Add `isSynchronized` field and constructor:
  ```java
  private boolean isSynchronized;

  public LockCountAnalysis(DepthFirstSearch dfs, boolean isSynchronized) {
    super(dfs);
    this.isSynchronized = isSynchronized;
  }
  ```

- Superclass constructor requires `DepthFirstSearch` of method `CFG`

- Fact at CFG entry will depend on whether or not method is synchronized
Fact creation, copying, equality comparison

```java
public LockCount createFact() {
    return new LockCount(0);
}

public void initEntryFact(LockCount result)
    throws DataflowAnalysisException {
    result.setCount(isSynchronized ? 1 : 0);
}

public void copy(LockCount source, LockCount dest) {
    dest.setCount(source.getCount());
}

public boolean same(LockCount fact1, LockCount fact2) {
    return fact1.getCount() == fact2.getCount();
}
```
Handling of Top and Bottom values:

```java
@Override
public boolean isFactValid(LockCount fact) {
   return !(fact.isTop() || fact.isBottom());
}

public boolean isTop(LockCount fact) {
   return fact.isTop();
}

public void makeFactTop(LockCount fact) {
   fact.setCount(LockCount.TOP);
}
```
LockCountAnalysis implementation (continued)

Instruction transfer function:

```java
@Override
public void transferInstruction(InstructionHandle handle, BasicBlock basicBlock, LockCount fact)
    throws DataflowAnalysisException {
    if (!isFactValid(fact)) {
        return;
    }

    short opcode = handle.getInstruction().getOpcode();
    if (opcode == Constants.MONITORENTER) {
        fact.setCount(fact.getCount() + 1);
    } else if (opcode == Constants.MONITOREXIT) {
        fact.setCount(fact.getCount() - 1);
    }

    if (fact.getCount() < 0) {
        fact.setCount(LockCount.BOTTOM);
    }
}
```
Meet function

```java
public void meetInto(LockCount fact, Edge edge, LockCount result)
    throws DataflowAnalysisException {
    if (fact.isTop() || result.isBottom()) {
        return;
    }
    if (fact.isBottom() || result.isTop()) {
        copy(fact, result);
        return;
    }
    if (fact.getCount() != result.getCount()) {
        result.setCount(LockCount.BOTTOM);
    }
}
```
LockCountDataflow

![New Java Class dialog box](image)

```
Source folder: demoplugin/src
Package: net.findbugs.demo.analysis
Enclosing type:
Name: LockCountDataflow
Modifiers: public
Superclass: s.findbugs.ba.Dataflow<LockCount, LockCountAnalysis>
Interfaces:
```

David Hovemeyer and William Pugh

Using FindBugs™ for Research
By convention, each DataflowAnalysis subclass has a corresponding Dataflow subclass.

```java
public class LockCountDataflow extends Dataflow<LockCount, LockCountAnalysis> {
    public LockCountDataflow(CFG cfg, LockCountAnalysis analysis) {
        super(cfg, analysis);
    }
}
```

LockCountDataflow will serve as the analysis object storing the results of performing LockCountAnalysis on a method.
LockCountDataflowEngine

- Implements IMethodAnalysisEngine
- Given a MethodDescriptor, produce a LockCountDataflow object for the method
- General process:
  1. Request any analysis objects needed; e.g., DepthFirstSearch
  2. Create LockCountAnalysis and LockCountDataflow objects
  3. Execute the dataflow algorithm
  4. Return the finished LockCountDataflow
LockCountDataflowEngine (continued)
Boilerplate methods:

- Register the engine to produce LockCountDataflow objects:
  ```java
  public void registerWith(IAnalysisCache analysisCache) {
    analysisCache.registerMethodAnalysisEngine(
        LockCountDataflow.class, this);
  }
  ```

- The analysis results can be recomputed; don’t need to retain indefinitely
  ```java
  public boolean can recompute() {
    return true;
  }
  ```
Perform the analysis and return the completed LockCountDataflow object:

```java
public Object analyze(IAnalysisCache analysisCache,
                       MethodDescriptor descriptor) throws CheckedAnalysisException {
  CFG cfg =
      analysisCache.getMethodAnalysis(CFG.class, descriptor);
  DepthFirstSearch dfs =
      analysisCache.getMethodAnalysis(DepthFirstSearch.class, descriptor);
  Method method =
      analysisCache.getMethodAnalysis(Method.class, descriptor);

  LockCountAnalysis analysis = new LockCountAnalysis(
      dfs, method.isSynchronized());
  LockCountDataflow dataflow = new LockCountDataflow(cfg, analysis);

  dataflow.execute();
  return dataflow;
}
```
Create EngineRegistrar class

![Java Class creation](image)

- **Source folder:** `demoplugin/src`
- **Package:** `net.findbugs.demo.analysis`
- **Name:** `EngineRegistrar`
- **Superclass:** `java.lang.Object`
- **Interfaces:** `md.cs.findbugs.classfile.IAnalysisEngineRegistrar`
Implementing the EngineRegistrar class

Create each kind of analysis engine supported by the plugin and register it with the analysis cache.

```java
public void registerAnalysisEngines(IAnalysisCache analysisCache) {
    new LockCountDataflowEngine().registerWith(analysisCache);
}
```
Add EngineRegistrar element to findbugs.xml

- Adding an EngineRegistrar element to findbugs.xml ensures that FindBugs will create an instance of the EngineRegistrar class and invoke its registerAnalyses method.

```xml
<EngineRegistrar
    class="net.findbugs.demo.analysis.EngineRegistrar"/>
```
Testing LockCountAnalysis/Dataflow

- Build/install plugin
- cd to bin/net/findbugs/demo/testcode subdirectory
- Execute command:
  ```
  findbugs -textui -visitors TestDataflowAnalysis \ 
  -Ddataflow.classname=net.findbugs.demo.analysis.LockCountDataflow \ 
  -Ddataflow.printcfg=true \ 
  TestISYNC.class
  ```

  Executes LockCountAnalysis/Dataflow on each method of specified test class, prints control flow graphs annotated with dataflow values
Create `FindInconsistentSync` detector class

![New Java Class dialog box]

- **Source folder**: `demoplugin/src`
- **Package**: `net.findbugs.demo.detect`
- **Name**: `FindInconsistentSync`
- **Modifiers**: `public`
- **Superclass**: `edu.umd.cs.findbugs.bcel.CFGDetector`
FindInconsistentSync implementation

- For each analyzed method, iterate over Locations in method
- For each field, record synchronized and unsynchronized accesses
- At end of analysis pass, report fields with both synchronized and unsynchronized accesses
FindInconsistentSync implementation (continued)

static class AccessCount {
    int numSyncAccesses;
    int numUnsyncAccesses;
    List<SourceLineAnnotation> unsyncLocList =
        new LinkedList<SourceLineAnnotation>();

    boolean isInconsistent() {
        return numSyncAccesses > 0 && numUnsyncAccesses > 0;
    }

    public void addSyncAccess() {
        numSyncAccesses++;
    }

    void addUnsyncAccess(SourceLineAnnotation sourceLoc) {
        numUnsyncAccesses++;
        unsyncLocList.add(sourceLoc);
    }
}
private BugReporter bugReporter;
private Map<XField, AccessCount> accessMap;

public FindInconsistentSync(BugReporter bugReporter) {
    this.bugReporter = bugReporter;
    this.accessMap = new HashMap<XField, AccessCount>();
}

private AccessCount getAccessCount(XField field) {
    AccessCount accessCount = accessMap.get(field);
    if (accessCount == null) {
        accessCount = new AccessCount();
        accessMap.put(field, accessCount);
    }
    return accessCount;
}
@Override
protected void visitMethodCFG(MethodDescriptor methodDescriptor, CFG cfg) throws CheckedAnalysisException {
    IAnalysisCache analysisCache = Global.getAnalysisCache();

    ConstantPoolGen cpg = analysisCache.getClassAnalysis(
        ConstantPoolGen.class, methodDescriptor.getClassDescriptor());
    LockCountDataflow lockCountDataflow = analysisCache.getMethodAnalysis(
        LockCountDataflow.class, methodDescriptor);

    for (Iterator<Location> i = cfg.locationIterator(); i.hasNext(); ) {
        Location location = i.next();
        short opcode = location.getHandle().getInstruction().getOpcode();
        if (opcode != Constants.GETFIELD && opcode != Constants.PUTFIELD) {
            continue;
        }

        inspectFieldAccess(methodDescriptor, cpg, lockCountDataflow, location);
    }
}
private void inspectFieldAccess(
    MethodDescriptor methodDescriptor, ConstantPoolGen cpg,
    LockCountDataflow lockCountDataflow, Location location)
throws DataflowAnalysisException {

    LockCount lockCount = lockCountDataflow.getFactAtLocation(location);
if (lockCount.getCount() < 0) { return; } 

    XField field = XFactory.createXField(
        (FieldInstruction) location.getHandle().getInstruction(), cpg);
    AccessCount accessCount = getAccessCount(field);

    if (lockCount.getCount() > 0) {
        accessCount.addSyncAccess();
    } else {
        accessCount.numUnsyncAccesses++;
        accessCount.addUnsyncAccess(
            SourceLineAnnotation.fromVisitedInstruction(methodDescriptor, location));
    }
}
@Override
public void finishPass() {
    Iterator<Map.Entry<XField, AccessCount>> i =
        accessMap.entrySet().iterator();
    while (i.hasNext()) {
        Map.Entry<XField, AccessCount> entry = i.next();

        if (!entry.getValue().isInconsistent()) { continue; }

        BugInstance warning = new BugInstance(
            this, "ISYNC_INCONSISTENT_SYNC", Priorities.NORMAL_PRIORITY);
        warning.addClass(entry.getKey().getClassName());
        warning.addField(entry.getKey());
        for (SourceLineAnnotation sourceLoc : entry.getValue().unsyncLocList) {
            warning.addSourceLine(sourceLoc).describe("SOURCE_LINE_UNSYNC_ACCESS");
        }

        bugReporter.reportBug(warning);
    }
}
Modify findbugs.xml

- Add Detector element:
  ```xml
  <Detector class="net.findbugs.demo.detect.FindInconsistentSync"
            reports="ISYNC_INCONSISTENT_SYNC"
            hidden="false"/>
  ```

- Add BugPattern element:
  ```xml
  <BugPattern abbrev="ISYNC" type="ISYNC_INCONSISTENT_SYNC"
                category="MT_CORRECTNESS"/>
  ```
Modify messages.xml

- Add BugPattern element:

```xml
<BugPattern type="ISYNC_INCONSISTENT_SYNC">
  <ShortDescription>Inconsistent synchronization of field</ShortDescription>
  <LongDescription>Inconsistent synchronization of field {1}</LongDescription>
  <Details>
    <![CDATA[
      <p>Both synchronized and unsynchronized accesses were observed for this field.</p>
    ]]>  
  </Details>
</BugPattern>
```

- Add BugCode element

```xml
<BugCode abbrev="ISYNC">Inconsistent synchronization</BugCode>
```
public class TestISYNC {
    Object field;
    Object lock = new Object();

    void set(Object value) {
        synchronized (lock) {
            field = value;
        }
    }

    Object get() {
        return field;
    }
}
Test detector

- Run `findbugs -textui -visitors \ FindInconsistentSync *.class`
- Output:

  M M ISYNC: Inconsistent synchronization of field net.findbugs.demo.testcode.TestISYNC.field  Unsynchronized access at TestISYNC.java:[line 20]
We have implemented two small but realistic detectors
  ...with some significant simplifying assumptions
To make them production quality, would need to:
  * Run on real software artifacts
  * See if they find interesting bugs
  * If too many false positives, implement heuristics to suppress them
Advanced dataflow topics
FrameDataflowAnalysis

- Base class for analyses that model values in local variables and operand stack
- Dataflow fact type is subclass of Frame
  - The values in the Frame are what you work with (they form the lattice of dataflow values)
- Your main concerns:
  - Effect of instructions and control edges on values (transferInstruction and edgeTransfer methods)
  - Meet function for values
Frame

- A stack frame containing slots (local vars and operand stack values)
- Each slot contains a value
- Your analysis operates on the values
- The Java type you specify for values can be immutable
- BCEL weirdness:
  - BCEL considers long and double values to occupy two slots
  - Suggested approach: just put the same dataflow value in both
AbstractFrameModelingVisitor

- A base visitor class for modeling the effect of instructions on a Frame
  - Makes implementing `transferInstruction()` relatively painless
- You only need to override visit methods for instructions that are significant for your analysis
  - Default implementations of visit methods push a “default value” onto the stack any time stack values are produced
  - You override `getDefaultValue()` method to specify what the default value should be
  - E.g.: for null pointer analysis, default value is “not null”
mergeValues()

- The `mergeValues()` method in `FrameDataflowAnalysis` is called to compute the meet of the values in a slot of two `Frame` objects, saving the result in the result frame.
- E.g.: for type analysis, meet of `java.math.BigInteger` and `java.math.BigDecimal` is `java.lang.Number`
FindBugs has a number of built-in dataflow analyses that may be useful:

- ValueNumberAnalysis/Dataflow
- TypeAnalysis/Dataflow
- DominatorsAnalysis/Dataflow
- LockAnalysis/Dataflow
ValueNumberAnalysis/Dataflow

ValueNumberDataflow vnaDataflow = analysisCache.getMethodAnalysis(
    ValueNumberDataflow.class, methodDescriptor);
...
ValueNumberFrame vnaFrame = vnaDataflow.getFactAtLocation(location);

- Answers question “do these two Frame slots definitely contain the same value?”
  - This is important because Frames often contain multiple copies of same value
    - E.g., after copying a value from local var to operand stack, dup instruction, etc.
  - Information gained about one value should be propagated to all other copies of same value
- Similar to SSA
TypeAnalysis/Dataflow

TypeDataflow typeDataflow = analysisCache.getMethodAnalysis(
    TypeDataflow.class, methodDescriptor);
...
TypeFrame typeFrame = typeDataflow.getFactAtLocation(location);

- Answers question “what type of value is stored in the Frame slot?”
- Values are instances of BCEL Type class
- Type-inference similar to JVM bytecode verifier
D dominatorsAnalysis = analysisCache.getMethodAnalysis(D dominatorsAnalysis.class, methodDescriptor);

...  BitSet dominators = dominatorsAnalysis.getStartFact(basicBlock);

- Get bitset of (integer labels of) BasicBlocks which dominate a particular BasicBlock
- Also:
  - NonExceptionPostDominatorsAnalysis
    - Ignores all exception control edges
  - NonImplicitExceptionPostDominatorsAnalysis
    - Ignores all “implicit” exception edges (e.g., undeclared runtime exceptions)
LockAnalysis/Dataflow

LockDataflow lockDataflow = analysisCache.getMethodAnalysis(LockDataflow.class, methodDescriptor);
...
LockSet lockSet = lockDataflow.getFactAtLocation(location);

- Set of ValueNumbers which have been locked using monitorenter
  - Counts number of times locked
- Intra-procedural
Interprocedural analysis
Interprocedural analysis

- FindBugs has some basic support for interprocedural analysis
- Recommended approach: compute method summaries in earlier analysis pass, use them in later pass
  - No real support for iterative analysis
  - Can fake it by saving information, reloading on next run
- FindBugs orders analysis of classes in pass by performing a topological sort on call graph
  - Try to analyze referenced class before class that references it
  - Data computed earlier in pass can be used later in same pass
Implementing an interprocedural analysis

- Use two detectors:
  - First pass detector computes method summaries
    - Guaranteed to see all classes referenced by application
    - It should implement the NonReportingDetector interface
  - Second pass detector uses computed method summaries
    - Issue: looking up method summary/summaries at call sites

- Use detector ordering constraints to ensure the two detectors run in separate analysis passes
Using a database to convey information between passes

- The first time the first-pass detector visits a class, create a database to store interprocedural information:

  ```java
  if (!createdDatabase) {
    MyDatabase db = new MyDatabase();
    analysisCache.eagerlyPutDatabase(MyDatabase.class, db);
    createdDatabase = true;
  }
  
  Getting the database (in first-pass or second-pass detector):

  MyDatabase db =
  analysisCache.getDatabase(MyDatabase.class);
  ```
Assume the first-pass detector extends CFGDetector:

```java
protected void visitMethodCFG(MethodDescriptor methodDescriptor, CFG cfg) {
    IAnalysisCache analysisCache = Global.getAnalysisCache();

    // Analyze method, compute summary
    MyDatabase db = analysisCache.getDatabase(MyDatabase.class);
    db.putSummary(methodDescriptor, summaryObject);
}
```
Using information from database in second pass

ConstantPoolGen cpg = analysisCache.getClassAnalysis(
    ConstantPoolGen.class, methodDescriptor.getClassDescriptor());
MyDatabase db = analysisCache.getDatabase(MyDatabase.class);

...

// Analyzing a call site
Instruction ins = location.getHandle().getInstruction();
if (ins instanceof InvokeInstruction) {
    InvokeInstruction inv = (InvokeInstruction) ins;
    MethodDescriptor calledMethod = BCELUtil.getCalledMethodDescriptor(inv, cpg);
    summary = db.getSummary(calledMethod);
    Use method summary...
}
Virtual call sites

- Database should combine summaries of all methods that implement or override the called method
  - Can do this lazily as call sites are encountered
- Use Hierarchy class to perform class hierarchy queries
  - I.e., which methods implement or override a given method
- In the future:
  - Would be nice to have a more general framework in place for analyzing virtual call sites
  - Possibly using a call graph construction algorithm (e.g., RTA) to prune possible call targets
Introduction

Start with a bug

Implementing detectors and analyses

Tracking warnings

Tools
Tracking warnings

Need to be able to identify when a warning of the same issue appears in the results of two different analyses [1]

- Same analyzer run on two different builds
  - Identify which issues are new
  - Allow audits and comments to be persistent (e.g., “Not a problem”, or “Joe should fix this”)

- Different builds of analyzer on the same software artifact
  - Everytime we change FindBugs, we want to know which reported warnings change
Tracking warnings is important

If you want any industrial adoption of static analysis, this is critically important.

- Tools general false positives or don’t care issues, don’t want to see those anew each time analysis is run.
- Persistence of audit/triage
- Manager feedback
- Tracking and improvement of tools
- Integration with bug tracker
Issues when tracking warnings

- Needs to be insensitive to unimportant changes: line numbers, indentation, etc.
- Some issues are challenging: package/class renaming, method refactoring, etc.
- Want to avoid accidental collisions (e.g., fix one null pointer bug but introduce another)
- Generally, practitioners would prefer to accidental collisions rather than seeing issues reappear
FindBugs supports two ways to track warnings

- Compare two bug collections, identify matching instances (each is matched to at most one instance in the other collection)
  - This approach does better fuzzy matching, can handle things such as package renaming
- Compute instance hash for each instance
  - This approach is more flexible, doesn’t assume linear sequence of versions, integrates better with databases and other static analysis tools
Matching bug collections

- Use `edu.umd.cs.findbugs.workflow.Update`
- Invoke `mergeCollections`
  - `BugCollection origCollection`
  - `BugCollection newCollection`
  - `boolean copyDeadBugs` - should issues not present in new collection be excluded?
  - `boolean incrementalAnalysis` - might `newCollection` contain results for only a subset of what was examined in `origCollection`
How matching works

Uses a sequence of matchers

- A precise matcher
- A version insensitive matcher: ignores source line annotations and integer percentage annotations
- A fuzzy matcher: allows error reclassifications (e.g., a different kind of null pointer warning)
- A fuzzy matcher that allows package renaming
Things that are important

- Classes, methods, fields
- Variable names
- String constants
Adding important stuff to your bug instances

- Code being analyzed should be compiled with variable names (and source line numbers)
- If you write a detector, decorate bug instance with things you expect to be invariant, such as variable names
- Not only will this help the user, but it will be useful in tracking defects across versions
Call getInstanceHash() on a bugInstance
But the string returned is not guaranteed to be unique
Call computeBugHashes() on a SortedBugCollection
append getInstanceOccurrenceNum() and getInstanceOccurrenceMax() to get a key that is unique for this bug collection
Combining bugs to reduce collisions

- If you are getting lots of collisions, consider merging the instances that would collide into a single bug instance that references multiple source statements.
- This is also helping when auditing defect warnings
- Use a BugAccumulator to make this easier
Introduction

Start with a bug

Implementing detectors and analyses

Tracking warnings

Tools
New GUI

- Provides user designation and annotation support
- Highlights multiple source lines
- Use dragging to reorganize JTree
Command line tools

- We’ve got a lot of command line tools
  - some ant tasks, need to add more
  - but all the command lines tools can be invoked from within ant
- We need to build a bigger, better tool chain
  - we’re open source, we welcome contributions
    - Maven (contributed), Cruise control (?), ...
XML analysis results

- We use XML as the standard output from our analysis engine.
- XML analysis results can be filtered, processed, displayed in the GUI, annotated, converted to text or HTML.
- XML can be plain, or with messages:
  - with text/messages provides all the text to allow you to convert the XML into meaningful HTML without further FindBugs involvement.
- findbugs -textui -xml rt.jar >rt.xml
- run findbugs
  - using the test user interface, rather than the GUI
  - generate XML output, rather than one bug/line
    - also have emacs output mode
  - analyze all the classes in rt.jar, write the output to rt.xml
filterBugs

- filterBugs -priority H -category C rt.xml hc.xml
- Read the bugs in rt.xml, filter out just the high priority correctness bugs, and write them to hc.xml
convertXmlToText

- convertXmlToText hc.xml
  - convert to simple one bug/line format
- convertXmlToText -html:fancy.xsl
  - convert to html using fancy.xsl style sheet
listBugDatabaseInfo & setBugDatabaseInfo

Set information about the analysis

- name name this analysis/version
- time Give the timestamp for this analysis
- addSource add a source directory
- findSource find and add all relevant source directories
unionBugs

- combine results from analyzing disjoint classes into a single analysis file
  - don’t use this if the analysis files contain overlapping results
computeBugHistory

- computeBugHistory -output db.xml old.xml new.xml
  - combine the analysis results in old.xml and new.xml
  - write a historical analysis to db.xml
  - old.xml can be a historical analysis
matching old bugs with new bugs

- We do a number of clever things (or things we think are clever) to match warnings from an old analysis with warnings in a new analysis.
- Line numbers don’t matter.
- We err on overmatching.
  - if you modify a method, fixing one null pointer bug, and introducing another in the same method, we may think the bug hasn’t changed.
mineBugHistory

- mineBugHistory -formatDates -noTabs db.xml
  - produce a tabular listing of the number of bugs introduced and eliminated in each build/version in a historical analysis
Historical bug databases

- Each historical bug database records a sequence of versions/builds/analysis results
- Each analysis result has a name, a date and a sequence number (starting at 0)
Combing back to filterBugs

- filterBugs has lots of options
  filterBugs -first 1 db.xml | convertXmlToText
- filter out just the warnings that first appeared in sequence # 1 (the second analysis results), and convert the results to text
Importing bugs into your own bug databases

- if you want to bring our results into a database
  - generate xml with messages
  - use instance-hash
    - designed to be unique per bug, and match bugs across versions
    - Not as clever as the approach we use when matching XML, but still clever
Conclusions
Conclusions

- FindBugs can be a useful platform for bug-driven static analysis research
- Get involved!
  - Contributions gratefully accepted
  - findbugs-discuss@cs.umd.edu
References

J. Spacco, D. Hovemeyer, and W. Pugh.
Tracking defect warnings across versions.