COMP 5111 (2024Spring) WORKSHOP

Assignment 1 Introduction & Soot Tutorial

TA: Songqiang CHEN

Contact TA

Name: CHEN, Songqiang

Email: i9s.chen@connect.ust.hk

Office Hour: Rm 3663 (via Rm3661)
Mon & Wed 10:20 - 10:50

Zoom: Please send me an email to schedule a date and timeslot.

Assignment 1 Overview (10 pts)

- **Submission:** Canvas
- **Deadline:** 16 March 2024 (23:55)
- **Tasks:**
  - Task 1: Run Tools to Generate Test Cases and Measure Code Coverage (26%)
    - Task 1.1: To learn how to use **Randoop** for unit test generation
    - Task 1.2: To learn how to use **EclEmma** for code coverage measurement.
  - Task 2: Statement coverage measurement using **Soot** (25%)
  - Task 3: Branch coverage measurement using **Soot** (25%)
  - Task 4: Exploring the usefulness of **Generative AI** (24% + 10% bonus)

Detailed Tutorial

Detailed Specifications: https://github.com/CastleLab/COMP5111-2024Spring-Assignments-Students

Environment

- **OS: Linux** / MacOS / Windows
  - **Linux is highly recommended**
  - You can use the machine at CS lab.
- **JDK** 11
- **Eclipse** 2023.12
- **JUnit** 4.12, with hamcrest-1.3
- **Randoop** 4.3.1
- **Soot** 4.2.1

(May not be the latest, but should be the most stable.)

Task 1 (26%): Generate Test Cases & Measure Code Coverage

- Basically, task 1 only asks you to run existing software.
- You do not need to write code.
- All you need to do is following our instructions in our repository.

- Hints:
  - Make sure that you correctly install any related software and set appropriate environment variables.
  - Some students get error: *Class Not Found* – Check your Java classpath setup.

Task 1.1: Generate Tests with Randoop

- Objective:
  - Use Randoop to generate **FIVE** test **suites** for the program under test (named Subject.java).
- Steps:
  - ~~Install Randoop~~
  - Run Randoop to yield test cases.
  - Run the test cases for validation.
- Tutorials: Avail in our repository.

Random Testing with Randoop

Introduction

In this tutorial, you will learn how to use the random testing tool Randoop. Randoop is an automatic unit test generator for Java programs. It generates two types of test suites:

- Error-revealing tests that indicates errors in the Java code.
- Regression tests that capture current behavior of the program.

Steps

You can follow these steps to use Randoop to generate test cases.

1. Install Randoop

We have already prepared the target version: `randoop-all-4.3.1.jar` file in lib and you do not need to do anything on the installation.

In case you want to manually install the Randoop, please refer to the official guide of Randoop and choose the target version: `randoop-all-4.3.1.jar`

Please Note the subtle differences to set environment variables on Windows and Unix-based systems.

2. Generate Test Cases

After setting up the environment for Randoop, generating junit test cases is as easy as a single line of command.

First of all, you need to download assignment 1 project, make sure the file `src/main/java/comp5111/assignment/cut/Subject.java` exists.

Then, you need to compile the `Subject` class. (In case you are not familiar with Java, here is a tutorial to teach you how to use `java/javac` to compile the `.java` file to `.class` and how to configure the `classpath` when compiling)
Suppose the root dir of the project is `${ROOT_DIR}`, and the folder containing the java bytecode is `${TARGET}`, whose content should have `${TARGET}/comp5111/assignment/cut/Subject.class` file available.

To use Randoop to generate test cases for our class under test, you can use the following command:

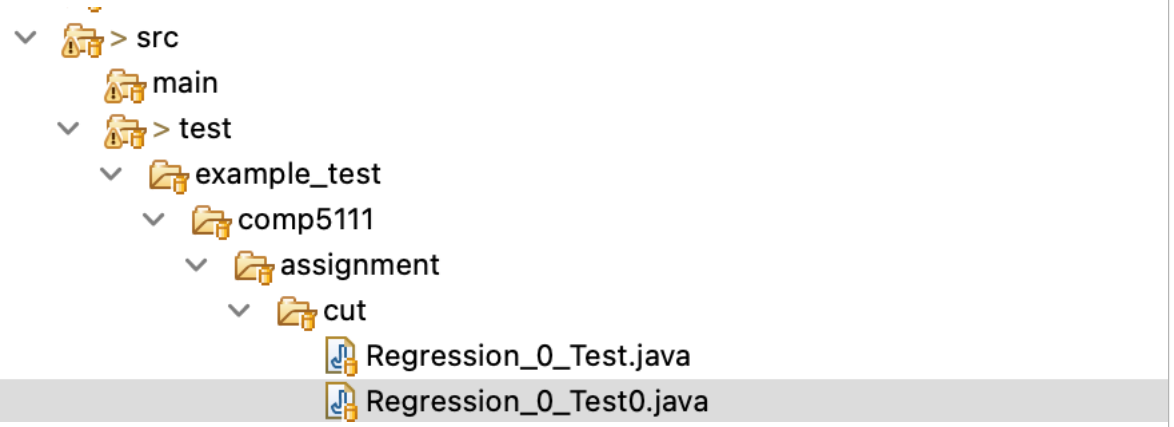
Linux and MacOs version (bash):

```
java -ea -classpath ${RANDOOP_JAR}:${TARGET}/ \
   randoop.main.Main gentests \
   --testclass=comp5111.assignment.cut.Subject \
   --time-limit=60 \
   --junit-package-name=comp5111.assignment.cut \
   --junit-output-dir=${ROOT_DIR}/src/test/randoop0
```

Task 1.1: Generate Tests with Randoop (cont.)

Note 2: Each test suite must achieve at least **40%** statement coverage!

- Output: test cases



Regression__Test.java | Regression__Test0.java

```
1  package comp5111.assignment.cut;
2
3  import org.junit.FixMethodOrder;
6
7  @FixMethodOrder(MethodSorters.NAME_ASCENDING)
8  public class Regression__Test0 {
9
10     public static boolean debug = false;
11
12     @Test
13     public void test01() throws Throwable {
14         if (debug)
15             System.out.format("%n%s%n", "Regression__Test0.test01");
16         char char0 = comp5111.assignment.cut.Subject.CharTasks.NUL;
17         org.junit.Assert.assertTrue("'" + char0 + "' != '" + '\000' + "'", char0 == '\000');
18     }
19
20     @Test
21     public void test02() throws Throwable {
22         if (debug)
23             System.out.format("%n%s%n", "Regression__Test0.test02");
24         int int0 = comp5111.assignment.cut.Subject.StringTasks.INDEX_NOT_FOUND;
25         org.junit.Assert.assertTrue("'" + int0 + "' != '" + (-1) + "'", int0 == (-1));
26     }
27
28     @Test
29     public void test03() throws Throwable {
30         if (debug)
31             System.out.format("%n%s%n", "Regression__Test0.test03");
32         boolean boolean1 = comp5111.assignment.cut.Subject.NumberTasks.isParsable("");
33         org.junit.Assert.assertTrue("'" + boolean1 + "' != '" + false + "'", boolean1 == false);
34     }
35
36     @Test
37     public void test04() throws Throwable {
38         if (debug)
39             System.out.format("%n%s%n", "Regression__Test0.test04");
40         java.lang.String str1 = comp5111.assignment.cut.Subject.StringTasks.chop("hi!");
41         org.junit.Assert.assertEquals("'" + str1 + "' != '" + "hi" + "'", str1, "hi");
42     }
43
44     @Test
45     public void test05() throws Throwable {
46         if (debug)
47             System.out.format("%n%s%n", "Regression__Test0.test05");
48         java.lang.String str1 = comp5111.assignment.cut.Subject.FilenameTasks.getName("hi!");
49         org.junit.Assert.assertEquals("'" + str1 + "' != '" + "hi!" + "'", str1, "hi!");
50     }
51
```

Note 1: Randoop is designed to generate **regression tests** that capture **current behavior** of program. **Error-revealing tests** that indicates **error** (e.g., crash and unexpected NPE) in the program under test are **not required** in your submitted generated test suites.

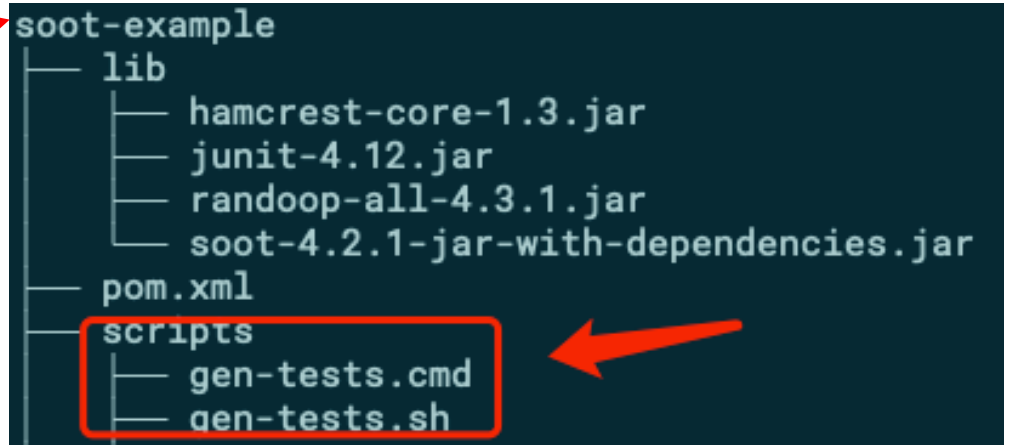
Task 1.1: Generate Tests with Randoop (cont.)

We have already prepared the example script for you to run Randoop and generate tests.

Programming Assignments

Workshop: Assignment 1 Introduction and Soot Tutorial (Feb 14)

slides soot example project




```
java -classpath .:"$ROOT_DIR"/raw-classes:"$ROOT_DIR"/lib/* randoop.main.Main \
  gentests --testclass castle.comp5111.example.Subject --output-limit 50 \
  --junit-output-dir "$DIR"/../src/main/java --junit-package-name castle.comp5111.example.test
```

Task 1.2: Measuring Test Coverage using EclEmma (in Eclipse)

- Objective:
  - Use Eclemma to measure the coverage of generated test cases.
- Steps:
  - Install **EclEmma / Eclipse**.
  - Create the Java project.
  - Run the unit test cases.
  - Read and check the report.
- Tutorials: Avail in our repository.
  A demo video is also avail.

https://drive.google.com/file/d/1DZAGe0WpumKhFayyKqE2bDjnS3t9D42i/view?usp=sharing

Test coverage measurement using EclEmma on Eclipse

Introduction

In this tutorial, you will learn how to use the toolkit EclEmma in Eclipse to measure test coverage.

(Please note that some screenshots and demo video are materials of last year's course, they are only for this tutorial.)

Steps

You can follow these steps to build the JAVA project on Eclipse, then use it to generate test coverage reports. The demo video is available.

Step 0: Install Eclipse 2023-12 and EclEmma

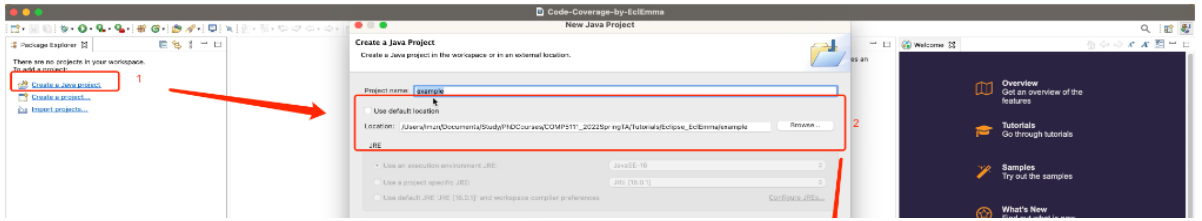
You can install the Eclipse 2023-12 following the official website If you have already installed the Eclipse with the old version, you may upgrade your Eclipse through this link

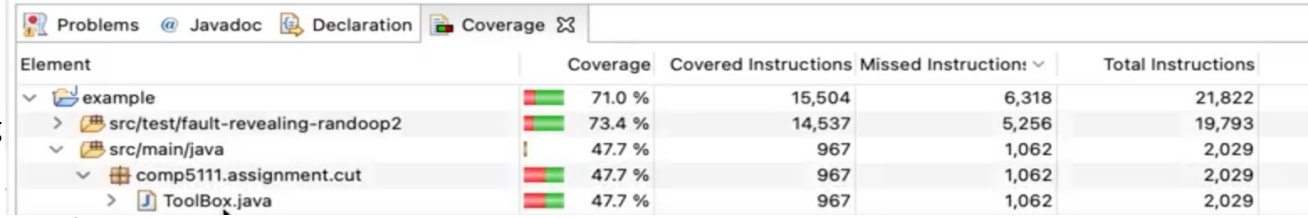
NOTE: EclEmma is already in Eclipse 2023-12 by default. You may skip this step.

Step 1: Create a JAVA project on Eclipse

As you have the test subject and the generated subject, you need to create a JAVA project from existing folders instead of building a JAVA project from scratch. Below shows you how to do so (suppose we want to create a JAVA project based on the following example folder)

Step 1-1: Create the new Java project by setting the existing location





Task 1 (26%): Generate Test Cases & Measure Code Coverage (cont.)

- **Submissions – Easy Enough ;)**
  - **Task 1.1:** 5 test suites generated by you.
  - **Task 1.2:** 10 screenshots. For each generated suite, 1 for statement coverage and 1 for branch coverage.

- **More Details on Format Requirement, Grading Scheme, etc.:**

  (All are concretely explained in the detailed instruction in our repository.)

Tasks 2 & 3 (25% + 25%): Measuring Test Coverage with Soot

- **Objective:** Develop a coverage measurement tool with Soot!
- It sounds hard, but actually, **it is easy**! - Once you understand Soot.
- First of all, you need to review the related coverage: (just for a rough recap :)
  - Instruction (Statement) Coverage
    - About the number of instructions that have been executed or missed in the program under test in one execution.
  - Branch Coverage
    - About the number of branches (e.g., branches of if) that have been executed or missed in the program under test in one execution.

Tasks 2 & 3 (25% + 25%): Measuring Test Coverage with Soot (cont.)

- To measure the coverage, you need to know:
  - How many statements (instructions) / branches in the program.
  - How many statements (instructions) / branches have been executed.
  - Coverage = $\frac{\#\ \boldsymbol{executed}\ statements(instructions)\ /\ branches}{\#\ \boldsymbol{total}\ statements(instructions)\ /\ branches}$

- To get answer for the above two "how many" questions, we need to ***instrument the program***!
  - Before execution, **we insert some code** to **ask** machine to record which stmt / branch has been executed, and output the record at the end of execution!
  - During execution, the inserted **code** will automatically **record** and **output** the runtime information.

Tasks 2 & 3 (25% + 25%) – Background Knowledge

- **Jimple 3-Address Code**: the intermediate code for Java used by Soot.
- Instead of directly handling *Java source code*, in PA, we will process the *Jimple 3AC* statements (instructions) and *Jimple If branches*.

```
Java:
...
for (i = 0; i < 10; ++i) {
    b[i] = i * i;
}
...
```



```
t1   := 0
L1: if t1 >= 10 goto L2
     t2 := t1 * t1
     t3 := t1 * 4
     t4 := b + t3
     *t4 := t2
     t1 := t1 + 1
     goto L1
L2:
```

```
; initialize i
; conditional jump
; square of i
; word-align address
; address to store i*i
; store through pointer
; increase i
; repeat loop
```

About 3AC:

- In 3AC, a Java expression is broken down into **several separate instructions**.
- There is at most one operator on the right side of an instruction
- Each 3AC contains at most 3 addresses

t2 = a + b + 3 ⇨ t1 = a + b
t2 = t1 + 3

Name a, b

Constant 3, “Hello”

Compiler-Generated Temporary t1, t2

Tasks 2 & 3 (25% + 25%) – Background Knowledge (cont.)

- **Instrument:** Insert some statements into the target program, in order to monitor some aspects of the program.
  - Must **not affect** the program behaviour.
  - May **affect** program runtime.
  - **Help us in:**
    - Code tracing
    - Debugging
    - Profiling
    - Data logging

```
t1   := 0
L1:  if t1 >= 10 goto L2
       t2 := t1 * t1
       t3 := t1 * 4
       t4 := b + t3
       *t4 := t2
       t1 := t1 + 1
       goto L1
L2:
```



```
t1   := 0
       (record a statement)
L1:  if t1 >= 10 goto L2
       (record a branch)
       (record a statement)
       t2 := t1 * t1
       (record a statement)
       t3 := t1 * 4
       (record a statement)
...
       goto L1
L2:    (record a branch)
```

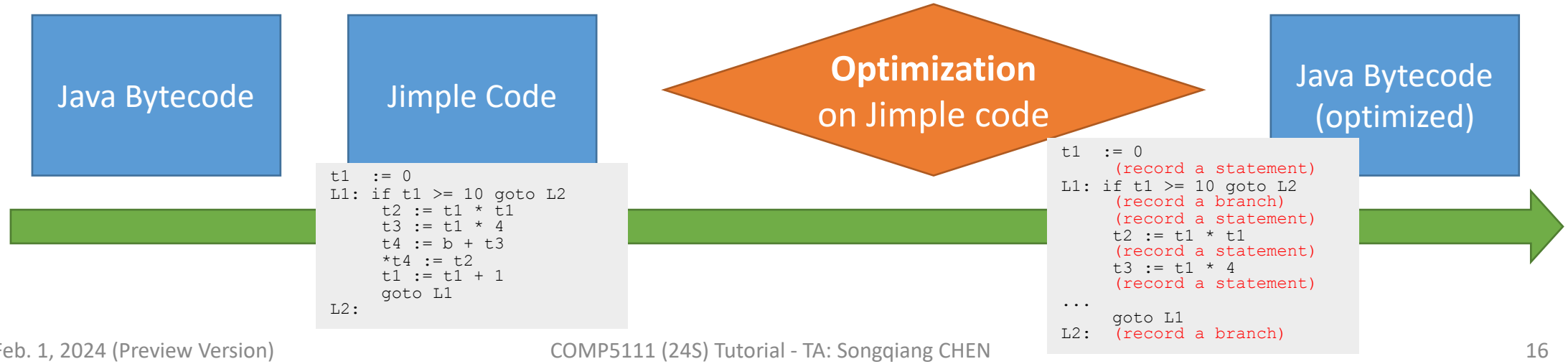
Tasks 2 & 3 (25% + 25%) – Soot

- We use a tool called Soot to instrument program.
  - Instead of manually inserting code.

- Soot: A Java Optimization Framework
  - Static Analysis (control flow analysis, call graph, point-to analysis, …)
  - Optimization & **Instrumentation** (automatically edit the code when traversing and analyzing the code.)

Tasks 2 & 3 (25% + 25%) – Soot (cont.)

- Soot: A Java Optimization Framework
  - Optimization & **Instrumentation** (automatically edit the code when traversing and analyzing the code.)
  - Input: Java Source Code / **Java Bytecode** (Recommended).
  - Intermediate Representation: **Jimple** (what we use) / Baf / Shimple / Grimp.
  - Output: Edited (optimized / instrumented) Java Bytecode.



Tasks 2 & 3 (25% + 25%) – Soot (cont.)

Try Soot and See What a Jimple Code Looks Like:

Foo.java

```java
public class Foo {

    static int factorial (int x){
        int result = 1;
        int i = 2;
        while (i <= x) {
            result *= i;
            i++;
        }
        return result;

    }
}
```

Foo.jimple

```
public class Foo extends java.lang.Object
{

    public void <init>()
    {
        Foo r0;

        r0 := @this: Foo;

        specialinvoke r0.<java.lang.Object: void <init>()>();

        return;
    }

    static int factorial(int)
    {
        int i0, i1, i2;

        i0 := @parameter0: int;

        i1 = 1;

        i2 = 2;

     label1:
        if i2 > i0 goto label2;

        i1 = i1 * i2;

        i2 = i2 + 1;

        goto label1;

     label2:
        return i1;
    }
}
```

Command

`java soot.Main Foo`

Process `Foo.class` in the current directory and produce a new class file in `sootOutput/Foo.class`.

`java soot.Main -f jimple Foo`

Same as above, but produce Jimple in `sootOutput/Foo.jimple`.

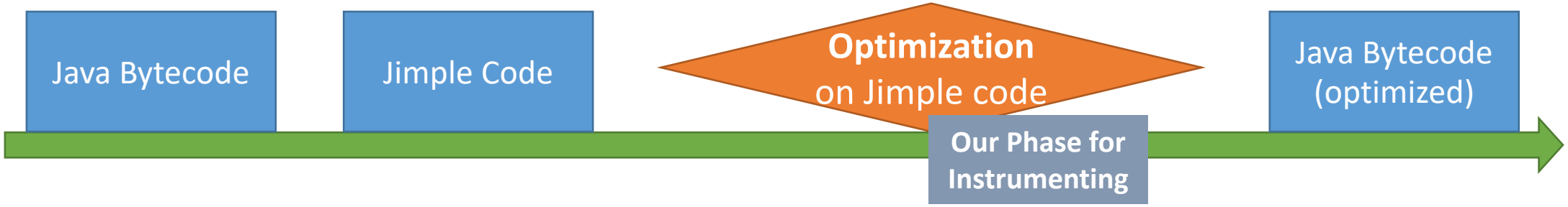
`java soot.Main -f dava Foo`

Decompile `Foo.class` and produce `Foo.java` in `sootOutput/dava/src/Foo.java`

To optimize Foo.jimple, we need some **customized optimization phases** for instrumentation.

Tasks 2 & 3 (25% + 25%) – Soot (cont.)

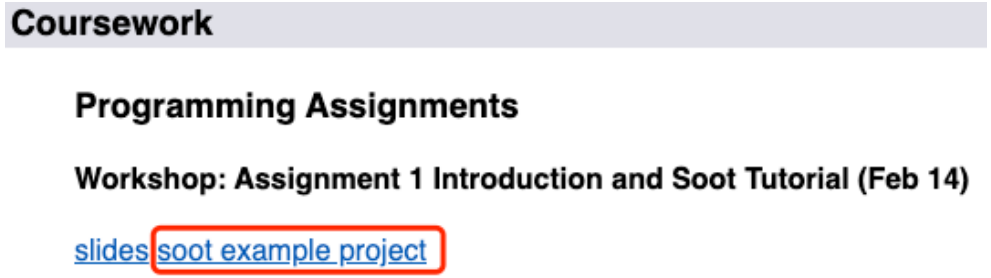
- The optimization could have multiple phases. Each phase can include a collection of transformers.
- We need to add our 'instrument' phase into the optimization, so that:
  - We will perform program instrumentation in our phase.
  - Our phase will process the program in the form of Jimple code (thus, our phase should be after the Jimple code is created).
  - Our phase will insert the instruction using Jimple code.



Tasks 2 & 3 (25% + 25%) – Example: Count Method Invocation

- **HOW? –** A Demo to Illustrate the Recording & Output Implementation.

  (download the project and try it before tutorial~)



- **Task:** Count *how many times* static / instance methods of the subject class (Class Under Test, CUT) are invoked by a test suites.

- First of all, consider **insert what code to where with what recorder**?
  – One sol: **Insert** a statement to **increase the counter in each method**.
- Then, create our own phase.

Tasks 2 & 3 (25% + 25%) – Example: Count Method Invocation (cont.)

Expected Outcome, Be Like …

```java
package castle.comp5111.example;

/**
 * This class is the subject in our tutorial of soot.
 * We first generate tests using randoop.
 * Then, we will use soot to instrument this class and count the total number of static/instance methods are executed
 * in tests.
 */
public class Subject {
    public static int staticMethod_Add(int x, int y) { return x + y; }

    public static double staticMethod_Add(double x, double y) { return x + y; }

    public static int staticMethod_Mul(int x, int y) { return x * y; }

    public static double staticMethod_Mul(double x, double y) { return x * y; }

    public int instanceMethod_Add(int x, int y) { return x + y; }

    public double instanceMethod_Add(double x, double y) { return x + y; }

    public int instanceMethod_Mul(int x, int y) { return x * y; }

    public double instanceMethod_Mul(double x, double y) { return x * y; }
}
```


Then,

- **Compile into a binary class.**
- **Execute the class with input to collect info.**
- **Output the record when the execution ends.**

Tasks 2 & 3 (25% + 25%) – Example – Our Own Phase – Counter

insert what code to where, with what recorder

- **How to record the info? –** Create a Helper Class: **Counter** (and call it when needed)

```
1   package castle.comp5111.example;
2
3   public class Counter {
4       private static int numStaticInvocations = 0;
5       private static int numInstanceInvocations = 0;
6
7       public static void addStaticInvocation(int n) { numStaticInvocations += n; }
10
11      public static void addInstanceInvocation(int n) {
12          numInstanceInvocations += n;
13      }
14
15      public static int getNumInstanceInvocations() { return numInstanceInvocations; }
18
19      public static int getNumStaticInvocations() { return numStaticInvocations; }
22  }
```

Counter Storage

The code to be inserted in CUT to record during CUT execution.

The code to be called after the execution of tests to get counting results.

Tasks 2 & 3 (25% + 25%) – Example – Our Own Phase – Transformer

insert what code to where, with what recorder

- **How to auto insert the caller code?** – A customized BodyTransformer (+soot).
  - Insert the **statement calling Counter methods before each return statement**.

```
public class Instrumenter extends BodyTransformer {

    /* some internal fields */
    static SootClass counterClass;
    static SootMethod addStaticInvocationMethod, addInstanceInvocationMethod;
    /*
     * internalTransform goes through a method body and inserts counter
     * instructions before method returns
     */
    @Override
    protected void internalTransform(Body body,
        // body's method
        SootMethod method = body.getMethod();

        // we dont instrument constructor (<init>) and static initializer (<clinit>)
        if (method.isConstructor() || method.isStaticInitializer()) {
            return;
        }

        // debugging
        System.out.println("instrumenting method: " + method.getSignature());

        // get body's unit as a chain
        Chain<Unit> units = body.getUnits();

        // get a snapshot iterator of the unit since we are going to
        // mutate the chain when iterating over it.
        //
        Iterator<?> stmtIt = units.snapshotIterator();
```

Key method: *internalTransform*

It will be executed when each method in the CUT is analyzed.

```
// typical while loop for iterating over each statement
while (stmtIt.hasNext()) {

    // cast back to a statement.
    Stmt stmt = (Stmt) stmtIt.next();

    // there are many kinds of statements, here we are only
    // interested in return statements
    // NOTE: there are two kinds of return statements, with or without return value
    if (stmt instanceof ReturnStmt || stmt instanceof ReturnVoidStmt) {
        // now we reach the real instruction
        // call Chain.insertBefore() to insert instructions
        //
        // 1. first, make a new invoke expression
        InvokeExpr incExpr = null;
        if (method.isStatic()) {
            // if current method is static, we add static method invocation counter
            incExpr = Jimple.v().newStaticInvokeExpr(
                    addStaticInvocationMethod.makeRef(), IntConstant.v(1));
        } else {
            // if current method is instance method, we add instance method invocation counter
            incExpr = Jimple.v().newStaticInvokeExpr(
                    addInstanceInvocationMethod.makeRef(), IntConstant.v(1));
        }

        // 2. then, make a invoke statement
        Stmt incStmt = Jimple.v().newInvokeStmt(incExpr);

        // 3. insert new statement into the chain, before return statement
        // (we are mutating the unit chain).
        units.insertBefore(incStmt, stmt);
    }
}
```

Tasks 2 & 3 (25% + 25%) – Example – Our Own Phase – Setup

- Setup for BodyTransformer:
  - Load the Helper Class Counter, so that we can:
  - Load the profiling methods, so that we can call them to record info in execution.
  - Load the report method, so that we can call it to print at the end of execution.

```
public class Instrumenter extends BodyTransformer {

    /* some internal fields */
    static SootClass counterClass;
    static SootMethod addStaticInvocationMethod, addInstanceInvocationMethod;

    static {
        counterClass = Scene.v().loadClassAndSupport( className: "castle.comp5111.example.Counter");
        addStaticInvocationMethod = counterClass.getMethod( subsignature: "void addStaticInvocation(int)");
        addInstanceInvocationMethod = counterClass.getMethod( subsignature: "void addInstanceInvocation(int)");
    }
```

Scene: a singleton class contains everything related to the code under analysis

Tasks 2 & 3 (25% + 25%) – Example – Our Own Phase – Setup (cont.)

- Setup for Soot:
  - Add our phase *Instrumenter* into Soot
    - After Jimple code is created!
  - Our own phase class `Instrumenter` should

    Extend an abstract class `BodyTransformer`

    `public class Instrumenter extends BodyTransformer{…}`

    Implement the method `internalTransform`

    `@override`

    `protected void internalTransform(Body body, …) {…}`
  - Our customized phase will be executed by:

```
/* Usage: java MainDriver [soot-options] appClass
import soot.*;

public class MainDriver {
    public static void main(String[] args) {

        /* check the arguments */
        if (args.length == 0) {
            System.err.println("Usage: java MainDriver [options] classname");
            System.exit(0);
        }

        /*Set the soot-classpath to include the helper class and class to analyze*/
        Options.v().set_soot_classpath(Scene.v().defaultClassPath()
            +File.pathSeparator+"bin/"+File.pathSeparator+"../Sample/bin/");

        /* add a phase to transformer pack by call Pack.add */
        Pack jtp = PackManager.v().getPack("jtp");
        jtp.add(new Transform("jtp.instrumenter", new InvokeStaticInstrumenter()));

        /*
         * Give control to Soot to process all options,
         * InvokeStaticInstrumenter.internalTransform will get called.
         */
        soot.Main.main(args);
    }
}
```

Tasks 2 & 3 (25% + 25%) – Prepare Runner Scripts

- Generate tests for CUT using randoop
  - scripts/gen-tests.sh for Linux / MacOS; scripts/gen-tests.cmd for Windows (**command line, CLI**)

```
for (( i = 0; i < 5; i++ )); do
  java -classpath "$RANDOOP_JAR:$ROOT_DIR/target/classes" randoop.main.Main gentests \
    --randomseed "$i" --testclass="comp5111.assignment.cut.Subject" --regression-test-basename="Regression_${i}_Test" \
    --junit-output-dir="$ROOT_DIR/src/test/randoop$i" --junit-package-name="comp5111.assignment.cut"
done
```

- Instrument and run tests
  - scripts/instrument-run-test.sh for Linux / MacOS; scripts/instrument-run-test.cmd for Windows (**CLI**)

```
for (( ti = 0; ti < 5; ti++ )); do
for (( i = 0; i < 2; i++ )); do
    arguments="$i comp5111.assignment.cut.Regression_"$ti"_Test "
    arguments=$arguments'comp5111.assignment.cut.Subject comp5111.assignment.cut.Subject$TaskHandler <…omit some classes…> '
    mvn clean compile exec:java -Dexec.mainClass="comp5111.assignment.Assignment1" \
    -Dexec.args="$arguments"
done
done
```

(Example codes here are for reference. They may be incomplete and contain minor errors.)

Tasks 2 & 3 (25% + 25%): Measuring Test Coverage with Soot (cont.)

Kind Tips 1/4 – Overall

- Start from the example project
  - Understand the code & logic about traversing statements and recording info.
  - Your assignment is *very similar* with the example.
- Go through the soot document
  - Learn other useful APIs besides the ones used in our toy example project!
- If you have problem:
  - Read FAQ in our repository first; contact with me (by email or open an issue).

Tasks 2 & 3 (25% + 25%): Measuring Test Coverage with Soot (cont.)

Kind Tips 2/4 – Where to Insert the Code

- Statement Coverage
  - Insert your new instructions for each target Jimple statement.

- Branch Coverage
  - Insert your new instructions for the branches under each `soot.jimple.IfStmt`;
  - Read FAQ in our repository first; contact with me (by email or open an issue).

Figure out **what logic & code** to insert by yourself.

Tasks 2 & 3 (25% + 25%): Measuring Test Coverage with Soot (cont.)

Kind Tips 3/4 – Necessary Hint

- Skip JIdentityStmt

3. Why we got the exception "java.lang.RuntimeException: @param-assignment statements should precede all non-identity statements" ?

This exception happened when you try to instrument the code before the declaration statement in Jimple Body. The Jimple body requires that, the declaration of the statement and assignments of parameters (including this object) should be always be in the front of other statements in the method body. For example:

```
public static void main(java.lang.String[])
{
    java.lang.String[] r0;
    int i0, $i1;
    java.io.PrintStream $r1;
    java.lang.StringBuilder $r2, $r3, $r4;
    java.lang.String $r5;
    r0 := @parameter0: java.lang.String[];
    i0 = 0;
    goto label1;
 label0:
    staticinvoke <TestInvoke: void foo()>();
    i0 = i0 + 1;
 label1:
    if i0 < 10 goto label0;
    $r1 = <java.lang.System: java.io.PrintStream out>;
    $r2 = new java.lang.StringBuilder;
    specialinvoke $r2.<java.lang.StringBuilder: void <init>(java.lang.String)>("I made ");
    $i1 = <TestInvoke: int calls>;
    $r3 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.StringBuilder append(int)>($i1);
    $r4 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>(" static calls'
    $r5 = virtualinvoke $r4.<java.lang.StringBuilder: java.lang.String toString()>();
    virtualinvoke $r1.<java.io.PrintStream: void println(java.lang.String)>($r5);
    return;
}
```

Therefore, please skip the instrumentation of these statements. We give a hint here, these statements are the objects of the class "soot.jimple.internal.JIdentityStmt".

- Crosscheck your implementation

For Tasks 2 and 3, you may take reference of an example test suite along with its statement coverage and branch coverage collected by our implementation using Soot. You can run your program on this example test suite and crosscheck the coverage results with ours, coverage reported by your tool should be close to ours.

COMP5111-2024Spring-Assignments-Students / src / test / example_test / TA-stmt.txt

imcsq (pub) Release Assignment 1. 3 hours ago

19 lines (13 loc) · 410 Bytes

Code Blame Raw

```
Overall:
percentage: 28.9%

================================================================

comp5111.assignment.cut.Subject$ParameterHandler
percentage: 17.1%

comp5111.assignment.cut.Subject$SortTools
percentage: 6.0%

comp5111.assignment.cut.Subject$NumberHandler
percentage: 38.9%

comp5111.assignment.cut.Subject$TaskHandler
percentage: 65.5%

comp5111.assignment.cut.Subject$TextHandler
percentage: 27.6%
```

Tasks 2 & 3 (25% + 25%): Measuring Test Coverage with Soot (cont.)

Kind Tips 4/4 – 2 Common Problems

- **The total number of statements recorded by Soot is different from EclEmma?**
  - **Normal** because EclEmma and Soot use different intermediate representation (IR). Thus, the total count of statements is different. You may only refer to the EclEmma stmt (branch) coverage for reference - they are close.

- **How to present the coverage of each statement / branch?**
  - You can choose any format you like, as long as the result is clear.
    Example:

```
if i0 < 10 goto label0;, yes
$r1 = <java.lang.System: java.io.PrintStream out>;, no
$r2 = new java.lang.StringBuilder;, no
specialinvoke $r2.<java.lang.StringBuilder: void (java.lang.String)>("I made ");, no
$i1 = <TestInvoke: int calls>;, no
```

Tasks 2 & 3 (25% + 25%): Useful Resources

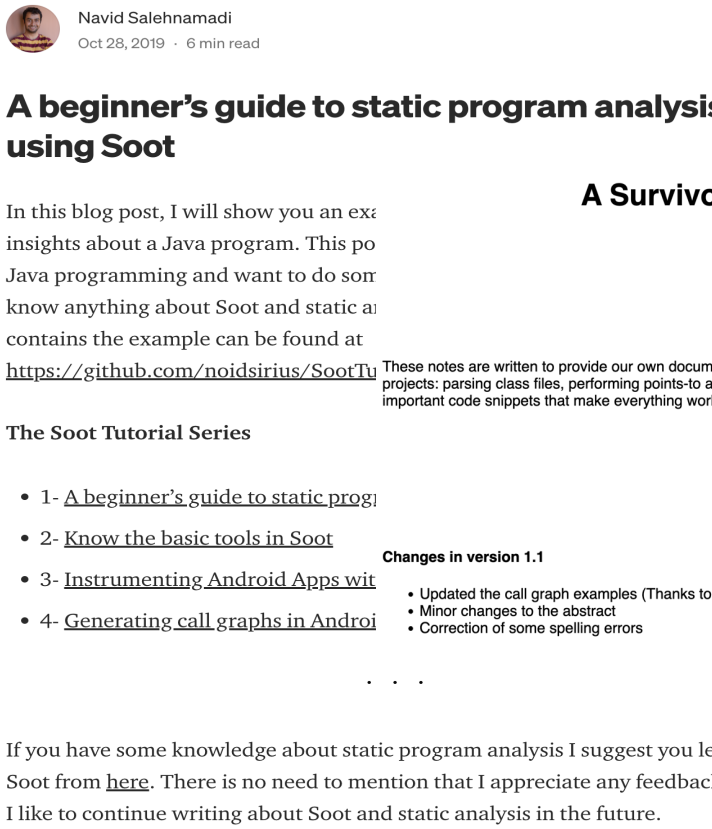
Soot Documentation: https://www.sable.mcgill.ca/soot/doc/overview-summary.html

Overview Package Class Use Tree Deprecated Index Help

PREV NEXT FRAMES NO FRAMES All Classes

Packages

soot	Base Soot classes, shared by different intermediate representations.
soot.baf	Public classes for the Baf intermediate representation.
soot.baf.internal	Internal, messy, implementation-specific classes for the Baf intermediate representation.
soot.baf.toolkits.base	A toolkit to optimize the Baf IR.
soot.coffi	Contains classes from the Coffi tool, by Clark Verbrugge.
soot.dava	
soot.dava.internal.asg	
soot.dava.internal.AST	
soot.dava.internal.javaRep	
soot.dava.internal.SET	
soot.dava.toolkits.base.AST	
soot.dava.toolkits.base.AST.analysis	
soot.dava.toolkits.base.AST.interProcedural	
soot.dava.toolkits.base.AST.structuredAnalysis	
soot.dava.toolkits.base.AST.transformations	
soot.dava.toolkits.base.AST.traversals	
soot.dava.toolkits.base.DavaMonitor	
soot.dava.toolkits.base.finders	
soot.dava.toolkits.base.misc	
soot.dava.toolkits.base.renamer	
soot.grimp	Public classes for the Grimp intermediate representation.
soot.grimp.internal	Internal, messy, implementation-specific classes for the Grimp intermediate representation.
soot.grimp.toolkits.base	A toolkit to optimize the Grimp IR.
soot.javaToJimple	
soot.javaToJimple.jj	Jjeton language extension.
soot.javaToJimple.jj.ast	AST nodes for the jjeton language extension.
soot.javaToJimple.jj.types	Type objects for the jjeton language extension.
soot.javaToJimple.toolkits	
soot.jbco	
soot.jbco.bafTransformations	
soot.jbco.gui	
soot.jbco.jimpleTransformations	
soot.jbco.util	



Navid Salehnamadi

Oct 28, 2019 · 6 min read

A beginner's guide to static program analysis using Soot

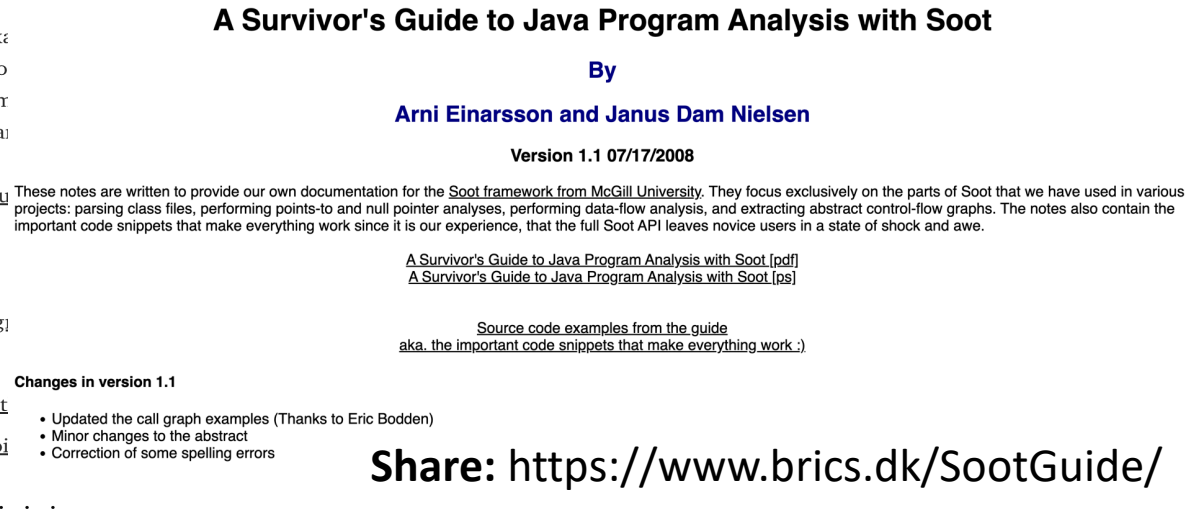
In this blog post, I will show you an exa
insights about a Java program. This po
Java programming and want to do som
know anything about Soot and static a
contains the example can be found at
https://github.com/noidsirius/SootTu

The Soot Tutorial Series

- 1- A beginner's guide to static prog
- 2- Know the basic tools in Soot
- 3- Instrumenting Android Apps wit
- 4- Generating call graphs in Androi

. . .

If you have some knowledge about static program analysis I suggest you learn Soot from here. There is no need to mention that I appreciate any feedback since I like to continue writing about Soot and static analysis in the future.



A Survivor's Guide to Java Program Analysis with Soot

By

Arni Einarsson and Janus Dam Nielsen

Version 1.1 07/17/2008

These notes are written to provide our own documentation for the Soot framework from McGill University. They focus exclusively on the parts of Soot that we have used in various projects: parsing class files, performing points-to and null pointer analyses, performing data-flow analysis, and extracting abstract control-flow graphs. The notes also contain the important code snippets that make everything work since it is our experience, that the full Soot API leaves novice users in a state of shock and awe.

A Survivor's Guide to Java Program Analysis with Soot [pdf]
A Survivor's Guide to Java Program Analysis with Soot [ps]

Source code examples from the guide
aka. the important code snippets that make everything work :)

Changes in version 1.1

- Updated the call graph examples (Thanks to Eric Bodden)
- Minor changes to the abstract
- Correction of some spelling errors

Share: https://www.brics.dk/SootGuide/

Blog: https://noidsirius.medium.com/a-beginners-guide-to-static-program-analysis-using-soot-5aee14a878d

Task 4 (24% + Bonus 10%): Explore the Usefulness of GenAI

- Requirements: (Please refer to our very detailed instructions)
  - Explore the usefulness of GenAI (LLMs like GPT) in previous 3 tasks (test generation & tasks on branch/stmt coverage).
  - Write a report to show an effective prompt(s) & findings (discussing effectiveness of LLM) **for each of** Tasks 1-3.

    *3 * (3% prompt + 5% finding). Insightful findings are expected ~*

- Exploration Subjects: (c.f. the detailed instructions in repo)
  - T1: Ask LLM to generate effective tests;
  - T2/3: Ask LLM to estimate the coverage of test cases;
  - T2&3: Ask LLM to enhance coverage of existing tests;
  - All: Compare the performance of multiple LLMs.
  - (Bonus: Give advice on designing effective prompts.)

- Baseline Prompt (for reference):

```
[Task 1]

Prompt Strategy: I use 2 prompts in a chain of thoughts.
I: Generate five diverse failing unit tests with assert statements for the following function
LLM: xxxxxx.
I: Explain and validate the assertion in your generated tests. Revise it if you find it not g
LLM: yyyyyy.

Findings:
I found HKUST GPT3.5 effective for test generation when appropriate information about P and Q
The 3 tests generated by HKUST GPT3.5 are XXX, YYY, ZZZ.
Compilable tests: XXX, YYY. (The issue of ZZZ is ...)
Test with a correct assertion: XXX. (The issue of YYY is ...)

Advice:
Developers should contain info on A, B, and C in the prompt for compilable/effective tests. F
CoT is also found useful. For example, ...

[Task 2]
Prompt Strategy: I use 1 zero-shot prompt, which I found effective enough.
What is the statement coverage of applying the following test test003 on function getIndex?
Code:
public int getIndex(final String exp) {<omit code>}
public void test003() throws Throwable {<omit code>}
...(omit)...

[Task 3]
Prompt Strategy: I use 1 zero-shot prompt, which I found effective enough.
What is the branch coverage of applying the following test test003 on function getIndex?
Code:
public int getIndex(final String exp) {<omit code>}
public void test003() throws Throwable {<omit code>}
...(omit)...
```

Submission:

- **Submission Requirement:**
  - Submit via **Canvas**.
  - Please check our course website & repo for the very detailed specification.
    - all info mentioned in this tutorial, as well as the detailed submission format requirement, hint and guidance, FAQs, etc.

- **Deadline:** 23:55, 16th March 2024

Do it as soon as possible & Try your best.