Annotating a class created from scratch with Soot

Chris Goard (cgoard@sable.mcgill.ca)

February 10, 2005

This tutorial will show you how to add various kinds of class file attributes to a program, created from scratch, with Soot. It extends the program introduced in "Creating a class file from scratch with Soot." Another, slightly more detailed, illustration of this process is available in the Soot tutorial "Adding attributes to class files via Soot."

1   Class file Attributes

Attributes are name-value pairs that can be associated with various class file structures. The Java VM spec (chapter 4.7) defines four different kinds of attribute. (Or, to be more precise, attributes can be attached to four different class file structures.) The four are: class, field, method, and code.

All attributes have the following structure:

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

We will extend the program from the "Creating a class file from scratch with Soot" tutorial to add attributes of each kind to the class file it produces.

2   Soot Hosts and Tags

We can attach arbitrary metadata to a variety of Soot data structures. Some such metadata may be converted into class file attributes when a class file is produced, and some may be for internal Soot use only. Two interfaces, in the `soot.tagkit` package, define this metadata facility: `Host` and `Tag`. A `Tag` is a named piece of metadata and a `Host` is an object that may have any number of uniquely named `Tag`s attached.

If we wish to define a `Tag` that will be converted to a class file attribute, we use the `Attribute` subinterface and attach it to a soot structure that corresponds to one of the four class file structures to which attributes can be attached.

3   Adding Class and Method Attributes

In this section, we will extend our program to add class and method attributes to the resultant `HelloWorld` class. To add class and method attributes, we will use the `GenericAttribute` class, which is a default implementation of `Attribute` suitable for simple class, method and field attributes.

The following code snippets show how to add class and method attributes. The class attribute will have the value "foo" and the method attribute will have no data. The generated class file will now have these attributes.

First, we create our class attribute and add it to the `SootClass`. In the resultant class file, the `attribute_info` structure's `attribute_name_index` field will point to the Unicode string "ca.mcgill.sable-.MyClassAttr" in the constant pool, and the info field will contain the bytes of the Unicode string "foo".

```
// create and add the class attribute, with data ''foo''
   GenericAttribute classAttr = new GenericAttribute(
       "ca.mcgill.sable.MyClassAttr",
       "foo".getBytes());
   sClass.addTag(classAttr);
```

We do basically the same thing for a method attribute:

```
// Create and add the method attribute with no data
   GenericAttribute mAttr = new GenericAttribute(
       "ca.mcgill.sable.MyMethodAttr",
       "".getBytes());
   method.addTag(mAttr);
```

If the `HelloWorld` class being produced by this program had any fields, we could similarly add an attribute to a `SootField`.

4 Adding Code Attributes

According to section 4.7.3 of the Java VM Spec, every method structure in a class file will have exactly one `Code_attribute`, unless the method is native or abstract, in which case it will have zero. The `Code_attribute`, which is where the bytecode instructions of a method are stored, may have an arbitrary number of attributes attached. When we refer to "code attributes" in this tutorial, it is these optional attributes attached to the class file `Code_attribute` structure to which we refer. They have the same structure as class, method, and field attributes.

In general, we use code attributes to annotate bytecode instructions in the body of a method, since it is not possible to annotate instructions directly. For example, the `LineNumberTable` attribute annotates bytecode instructions with the source file line numbers of their source Java statements. It is usual that the value of a code attribute be the encoding of a table associating bytecode offsets to values.

The Soot structure that corresponds to the class file's `Code_attribute` structure is the `Body` class. Thus in order to add a code attribute to the resultant class file, we must add an `Attribute`, in Soot, to the method's `Body`. However, for several reasons it would be very inconvenient to implement instruction tagging in Soot by directly manipulating an attribute attached to a method's `Body`. Instead, Soot allows us to tag the instructions themselves, and provides facilities for automatically converting these instruction tags into a single code attribute when the class file is created.

4.1 Tagging Units in Soot

Soot bodies can be in one of several different intermediate representations. In our program, the body we are creating is populated with Jimple statements. Statements in each intermediate representation subclass `Unit`, which implements `Host`, and so to each we may add a `Tag`. Since tags attached to such statements don't correspond directly to an attribute in the produced class file, but rather are converted later by Soot, we add `Tag`s not `Attribute`s.

Let us say we wish to add an attribute called "ca.mcgill.sable.MyTag" to each bytecode, and that the tag will have an integer value. First we define a new class representing the tag:

```
private class MyTag implements Tag {
```

```
    int value;

    public MyTag(int value) {
        this.value = value;
    }

    public String getName() {
        return "ca.mcgill.sable.MyTag";
    }

    // output the value as a 4-byte array
    public byte[] getValue() {
        ByteArrayOutputStream baos = new ByteArrayOutputStream(4);
        DataOutputStream dos = new DataOutputStream(baos);
        try {
            dos.writeInt(value);
            dos.flush();
        } catch(IOException e) {
            System.err.println(e);
            throw new RuntimeException(e);
        }
        return baos.toByteArray();
    }
}
```

Next, we add it to several Jimple statements in our program:

```
// add "l0 = @parameter0"
    tmpUnit = Jimple.v().newIdentityStmt(arg,
        Jimple.v().newParameterRef(
            ArrayType.v(RefType.v("java.lang.String"), 1), 0));
    tmpUnit.addTag(new MyTag(1));
    units.add(tmpUnit);

// insert "tmpRef.println("Hello world!")"
{
    SootMethod toCall = Scene.v().getMethod(
        "<java.io.PrintStream: void println(java.lang.String)>");
    tmpUnit = Jimple.v().newInvokeStmt(Jimple.v().newVirtualInvokeExpr(
        tmpRef, toCall.makeRef(), StringConstant.v("Hello world!")));
    tmpUnit.addTag(new MyTag(2));
    units.add(tmpUnit);
}
```

We now have tags attached to some of the statements in the method body. We will now look at how to convert these into a code attribute that can be written out with the class file.

4.2 Tag Aggregators

In order to convert the tags on statements into a code attribute that can be written out in the class file, we must define a `TagAggregator`. A `TagAggregator` is a Soot `BodyTransformer` that accepts a body with tagged instructions, and produces a body with an equivalent code attribute. Previously, we used the `GenericAttribute` class to represent the attribute structure written in the class file; in this section, we will

be using Soot's `CodeAttribute` class, which is a default implementation of a bytecode offset to value table. (To try to create an `Attribute` by hand with the same data would not be easy: dealing with the body in an intermediate representation as we are, we don't yet know what the resultant bytecode offsets will be. `CodeAttribute` takes care of this for us.)

A `TagAggregator` works by constructing a list of `Units` and a list of `Tags`. The list of `Units` denotes eventual bytecode offsets in the offset-value table, and the list of `Tags` the corresponding values. These lists must be the same length. We need to define three methods. The first, `wantTag`, is a predicate on tags that selects only those tags we are interested in. In our case, those that are instances of the `MyTag` class. The second, `considerTag`, populates the two lists. The third, `AggregatedName`, returns the name of the resultant attribute.

```
class MyTagAggregator extends TagAggregator {

    public String aggregatedName() {
        return "ca.mcgill.sable.MyTag";
    }

    public boolean wantTag(Tag t) {
        return (t instanceof MyTag);
    }

    public void considerTag(Tag t, Unit u) {
        units.add(u);
        tags.add(t);
    }
}
```

`TagAggregator` is an instance of `BodyTransformer`, and to use it we pass a method body to its transform method. However, the `TagAggregator` transform method expects a Baf body, and the body we are working with is currently a Jimple body. We must, therefore, first convert the body to Jimple. Each tag on each Jimple statement will be propagated to the corresponding Baf statements.

```
        MyTagAggregator mta = new MyTagAggregator();
        // convert the body to Baf
        method.setActiveBody(
            Baf.v().newBody((JimpleBody) method.getActiveBody()));
        // aggregate the tags and produce a CodeAttribute
        mta.transform(method.getActiveBody());
```

Since our method now has a Baf body, we must modify the code used to write the class out to a file:

```
        // write the class to a file
        String fileName = SourceLocator.v()
            .getFileNameFor(sClass, Options.output_format_class);
        OutputStream streamOut = new JasminOutputStream(
            new FileOutputStream(fileName));
        PrintWriter writerOut = new PrintWriter(
            new OutputStreamWriter(streamOut));
        AbstractJasminClass jasminClass = new soot.baf.JasminClass(sClass);
        jasminClass.print(writerOut);
        writerOut.flush();
        streamOut.close();
```

The resultant class file will now contain a code attribute that encodes a table mapping bytecode offsets to values.

4.3 CodeAttribute Binary Format

The binary format of the Soot `CodeAttribute` is as follows. The first two bytes are a count of entries in the table. Next is a stream of offset-value pairs. The offset is two bytes, and the value is of arbitrary size, the data being that returned by the `Tag`'s `getValue()` method.

Appendix A: Complete code for `createclass` example

The code for this example is reproduced below. It can be downloaded at:
   `http://www.sable.mcgill.ca/soot/tutorial/tagclass/Main.java`.

```
/* Soot - a J*va Optimization Framework
 * Copyright (C) 1997-1999 Raja Vallee-Rai
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the
 * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

/*
 * Modified by the Sable Research Group and others 1997-1999.
 * See the 'credits' file distributed with Soot for the complete list of
 * contributors.  (Soot is distributed at http://www.sable.mcgill.ca/soot)
 */


import soot.*;
import soot.baf.*;
import soot.jimple.*;
import soot.options.Options;
import soot.tagkit.*;
import soot.util.*;
import java.io.*;
import java.util.*;

/** Example of using Soot to create a classfile from scratch.
 * The 'createclass' example creates a HelloWorld class file using Soot.
 * It proceeds as follows:
 *
 * - Create a SootClass <code>HelloWorld</code> extending java.lang.Object.
 *
```

```
 * - Create a 'main' method and add it to the class.
 *
 * - Create an empty JimpleBody and add it to the 'main' method.
 *
 * - Add locals and statements to JimpleBody.
 *
 * - Write the result out to a class file.
 */

public class Main
{
    public static void main(String[] args) throws FileNotFoundException, IOException
    {
        SootClass sClass;
        SootMethod method;

        // Resolve dependencies
            Scene.v().loadClassAndSupport("java.lang.Object");
            Scene.v().loadClassAndSupport("java.lang.System");

        // Declare 'public class HelloWorld'
            sClass = new SootClass("HelloWorld", Modifier.PUBLIC);

        // 'extends Object'
            sClass.setSuperclass(Scene.v().getSootClass("java.lang.Object"));
            Scene.v().addClass(sClass);

        // create and add the class attribute
            GenericAttribute classAttr = new GenericAttribute(
                "ca.mcgill.sable.MyClassAttr",
                "foo".getBytes());
            sClass.addTag(classAttr);

        // Create the method, public static void main(String[])
            method = new SootMethod("main",
                    Arrays.asList(new Type[] {ArrayType.v(RefType.v("java.lang.String"), 1)}),
                    VoidType.v(), Modifier.PUBLIC | Modifier.STATIC);

            sClass.addMethod(method);

        // Create and add the method attribute
            GenericAttribute mAttr = new GenericAttribute(
                "ca.mcgill.sable.MyMethodAttr",
                "".getBytes());
            method.addTag(mAttr);

        // Create the method body
        {
             // create empty body
             JimpleBody body = Jimple.v().newBody(method);
```

```java
    method.setActiveBody(body);
    Chain units = body.getUnits();
    Local arg, tmpRef;
    Unit tmpUnit;

    // Add some locals, java.lang.String l0
        arg = Jimple.v().newLocal("l0", ArrayType.v(RefType.v("java.lang.String"), 1));
        body.getLocals().add(arg);

    // Add locals, java.io.printStream tmpRef
        tmpRef = Jimple.v().newLocal("tmpRef", RefType.v("java.io.PrintStream"));
        body.getLocals().add(tmpRef);

    // add "l0 = @parameter0"
        tmpUnit = Jimple.v().newIdentityStmt(arg,
            Jimple.v().newParameterRef(ArrayType.v(RefType.v("java.lang.String"), 1), 0));
        tmpUnit.addTag(new MyTag(1));
        units.add(tmpUnit);

    // add "tmpRef = java.lang.System.out"
        units.add(Jimple.v().newAssignStmt(tmpRef, Jimple.v().newStaticFieldRef(
            Scene.v().getField("<java.lang.System: java.io.PrintStream out>").makeRef())));

    // insert "tmpRef.println("Hello world!")"
    {
        SootMethod toCall = Scene.v().getMethod(
            "<java.io.PrintStream: void println(java.lang.String)>");
        tmpUnit = Jimple.v().newInvokeStmt(Jimple.v().newVirtualInvokeExpr(
            tmpRef,
            toCall.makeRef(),
            StringConstant.v("Hello world!")));
        tmpUnit.addTag(new MyTag(2));
        units.add(tmpUnit);
    }

    // insert "return"
        units.add(Jimple.v().newReturnVoidStmt());

}


MyTagAggregator mta = new MyTagAggregator();
// convert the body to Baf
method.setActiveBody(
    Baf.v().newBody((JimpleBody) method.getActiveBody()));
// aggregate the tags and produce a CodeAttribute
mta.transform(method.getActiveBody());

// write the class to a file
String fileName = SourceLocator.v().getFileNameFor(sClass, Options.output_format_class);
OutputStream streamOut = new JasminOutputStream(
                        new FileOutputStream(fileName));
```

7

```java
        PrintWriter writerOut = new PrintWriter(
                                    new OutputStreamWriter(streamOut));
        AbstractJasminClass jasminClass = new soot.baf.JasminClass(sClass);
        jasminClass.print(writerOut);
        writerOut.flush();
        streamOut.close();
    }
}

class MyTag implements Tag {

    int value;

    public MyTag(int value) {
        this.value = value;
    }

    public String getName() {
        return "ca.mcgill.sable.MyTag";
    }

    // output the value as a 4-byte array
    public byte[] getValue() {
        ByteArrayOutputStream baos = new ByteArrayOutputStream(4);
        DataOutputStream dos = new DataOutputStream(baos);
        try {
            dos.writeInt(value);
            dos.flush();
        } catch(IOException e) {
            System.err.println(e);
            throw new RuntimeException(e);
        }
        return baos.toByteArray();
    }
}


class MyTagAggregator extends TagAggregator {

    public String aggregatedName() {
        return "ca.mcgill.sable.MyTag";
    }

    public boolean wantTag(Tag t) {
        return (t instanceof MyTag);
    }

    public void considerTag(Tag t, Unit u) {
        units.add(u);
        tags.add(t);
    }
```

```
}
```

5 History

- February 10, 2005: Initial Version