Adding attributes to class files via Soot

Feng Qian (fqian@sable.mcgill.ca)
Patrick Lam (plam@sable.mcgill.ca)

June 13, 2002

Soot can annotate classfiles: for instance, it can add information about which array bounds checks and null pointer checks are redundant. We anticipate that users of Soot may wish to add new attributes to class files. This tutorial uses the array bounds check attribute to illustrate the internal structure of Soot annotation and describes how to add new attributes via Soot. Before reading this tutorial, readers should be familiar with the basic Soot classes, like SootClass, SootField, SootMethod, Body, and Unit. The other Soot tutorials explain these classes.

1 Structure of annotation

In release 1.2.0, we introduced a scheme for class file annotation in Soot. The general description of our annotation framework can be found in our CASCON paper, "A Framework for Optimizing Java Using Attributes". This tutorial explains practical issues related to implementing new attributes.

We first introduce the classes used for annotation. These classes reside in the `soot.tagkit` package.

Host interface

Any class which implements the `Host` interface promises that it can hold tags.

```
public interface Host
{
    /** Get a list of tags associated with the current object. */
    public List getTags();

    /** Returns a tag with the given name. */
    public Tag getTag(String aName);

    /** Adds a tag. */
    public void addTag(Tag t);

    /** Removes a tag with the given name. */
    public void removeTag(String name);

    /** Returns true if this host has a tag with the given name. */
    public boolean hasTag(String aName);
}
```

AbstractHost class

Soot provides a default implementation of the `Host` interface in the form of `AbstractHost`. Unless you have a pressing desire to provide the functionality yourself, any classes which you would like to implement `Host` should subclass `AbstractHost`. In Soot, the classes `SootClass`, `SootField`, `SootMethod`, `Body`, and `Unit` inherit from `AbstractHost`. Instances of these classes know how to carry tags, in the form of the `Tag` interface.

Tag interface

In Soot, we represent any annotation by an object whose type implements the `Tag` interface. This interface defines one methods: `getName()` returns the unique name of the tag (note that tag names must not conflict with each other).

```
public interface Tag
{
    /** Returns the tag's name. */
    public String getName();
}
```

Attribute interface

The `Attribute` interface extends `Tag`; it promises that the associated tag has attribute-like data which can be read and written as an array of bytes.

```
public interface Attribute extends Tag
{
    /** Returns the tag's raw data. */
    public byte[] getValue()
        throws AttributeValueException;

    /** Sets the value of the attribute from a byte[]. */
    public void setValue(byte[] v);
}
```

An `Tag` which is not an `Attribute` could be used to store arbitrary Soot information about a `Host`. An `Attribute` is something that would go in a classfile.

TagAggregator interface

The array-bounds check analysis annotates individual instructions as it discovers whether or not their bounds checks are required. More generally, analyses will attach attributes directly to the Units in question. However, the Java classfile structure does not make any provisions for directly attaching attributes to bytecodes, and attaching attributes directly to bytecodes would in any case be inefficient. Hence, we designed our attributes so that they would attach to a method in a tabular format: only one actual attribute is required per method body and tag type; this meta-attribute contains information about a number of different instructions.

An implementation of `TagAggregator` promises that it can combine all tags of some type into one big aggregated attribute, which can be attached to a method's code attribute. One implementation of a `TagAggregator` is `soot.jimple.toolkits.annotation.tags.ArrayNullTagAggregator`.

Aggregators can be active or inactive; the effect of being active or inactive is described later.

```
public interface TagAggregator
{
    /** Adds a new (unit, tag) pair. */
    public void aggregateTag(Tag t, Unit u);

    /** Generates the aggregated tag. */
    public Tag produceAggregateTag();

    /** Clears old accumulated tags. */
    public void refresh();
```

```
    /** Returns true if the aggregator is active. */
    public boolean isActive();
}
```

Base64 tool class

This utility class allows the encoding of raw bytes to base64-encoded characters and the decoding of base64 characters back to raw bytes.

JasminAttribute abstract class

Attributes are generated by analysis phases in the form of strings containing labels in the unit body and their values; for instance, we might have the attribute `"%label2%Aw==%label3%Ag==%label4%Ag=="` associated with a method body. In order to include this attribute in a class file, exact PC values are needed for the labels.

The `JasminAttribute` class provides a `decode` method which takes a string of (label, value) pairs and a map from labels to PCs and emits raw data, ready for inclusion in a class file. This method is called by Jasmin after the PC values are known. Any attribute which uses (label, value) pairs can subclass `JasminAttribute` to get output to classfiles for free; other attributes hoping to be output to classfiles must subclass `JasminAttribute` and override the `decode` method.

The abstract `getJasminValue()` method must return a string that can be included when outputting a `.jasmin` file. This string later gets decoded by `decode()`.

```
public abstract class JasminAttribute implements Attribute
{
    public static byte[] decode(String attr, Hashtable labelToPc);
    abstract public String getJasminValue(Map instToLabel);
}
```

CodeAttribute class

This class provides an implementation of the abstract `getJasminValue()` method of `JasminAttribute`. The `getJasminValue()` method must return a string reflecting the contents of its `CodeAttribute`. It may use the provided `instToLabel` map to convert `Unit`s into labels used in the returned `String`.

```
public class CodeAttribute extends JasminAttribute
{
    public String getJasminValue(Map instToLabel);
}
```

This type of attribute is clearly intended to be used for attributes associated with code.

GenericAttribute class

Java describes how three other types of attributes can be created: attributes may be associated with methods, fields and classes as well as code. Soot supports these attributes via the `GenericAttribute` class. Any such attribute can be created with an attribute name and a byte array value; it can then be attached to `SootClass`, `SootField`, or `SootMethod`.

```
public class GenericAttribute implements Attribute
{
    public GenericAttribute(String name, byte[] value);
    public String getName();
    public byte[] getValue();
}
```

The above classes provide APIs useful for adding new attributes. Soot attributes are represented as `Tag`s, and are attached to `Host`s. An exception is CodeAttribute. Because the tags for CodeAttribute are attached to units, a `TagAggregator` is used to combine them. Another class, `soot.baf.toolkits.base.CodeAttributeGenerator`, is a phase of Baf generation. This class calls some `TagAggregator`s and attaches the resulting aggregated tags to method bodies, corresponding to attributes on the code attribute. The `CodeAttributeGenerator` class has a method `registerAggregator`, which an aggregator can use to register callbacks. An aggregator can be set as active or inactive; during Baf generation, an active aggregator will accept tags according to the tag type.

In the array bounds check attribute example, a `ArrayNullTagAggregator` is registered by the `Main` class, accepting `ArrayCheckTag`s and `NullCheckTag`s.

```
CodeAttributeGenerator.v().registerAggregator(new ArrayNullTagAggregator(true));
```

2 Adding method attributes in Soot

Adding a code attribute is non-trivial, as it requires that an aggregator be provided. We first give a trivial example of adding a method attribute via `GenericAttribute`. The code can be found in `ashes.examples.addattributes`. It can also be downloaded at:

`http://www.sable.mcgill.ca/soot/tutorial/addattributes/Main.java`

We proceed by adding a new phase to the `jtp Pack`, called `annotexample`.

```
package ashes.examples.addattributes;

import soot.*;
import soot.tagkit.*;
import java.util.*;

public class AnnExample
{
    public static void main(String[] args)
    {
        /* adds the transformer. */
        Scene.v().getPack("jtp").add(new
                        Transform("annotexample",
                        AnnExampleWrapper.v()));

        /* invokes Soot */
        soot.Main.main(args);
    }
}
```

The `AnnExampleWrapper` is a subclass of `BodyTransformer`, which implements the `internalTransform` method. It simply adds a string "Hello world!" as an attribute to every method. The attribute has the name 'Example'.

```
public class AnnExampleWrapper extends BodyTransformer
{
    private static AnnExampleWrapper instance =
        new AnnExampleWrapper();

    private AnnExampleWrapper() {};

    public static AnnExampleWrapper v()
    {
```

```
        return instance;
    }

    public void internalTransform(Body body, String phaseName, Map options)
    {
        SootMethod method = body.getMethod();
        String attr = new String("Hello world!");

        Tag example = new GenericAttribute("Example", attr.getBytes());
        method.addTag(example);
    }
}
```

We recompile `foo` and annotate it with new attribute.
`java AnnExample foo`
The annotated class file has an "Example" attribute for each method. The string "Hello world!" is in binary form.

```
public class foo extends java.lang.Object
filename                foo
compiled from           foo.jasmin
compiler version        45.3
access flags            33
constant pool           14 entries
ACC_SUPER flag          true

Attribute(s):
        SourceFile(foo.jasmin)

2 methods:
        public void <init>() <(Unknown attribute Example:
                        48 65 6c 6c 6f 20 77 6f 72 6c 64 21)>
        void footest() <(Unknown attribute Example:
                        48 65 6c 6c 6f 20 77 6f 72 6c 64 21)>

public void <init>() <(Unknown attribute Example:
                        48 65 6c 6c 6f 20 77 6f 72 6c 64 21)>
Code(max_stack = 1, max_locals = 1, code_length = 5)
0:    aload_0
1:    invokespecial        java.lang.Object.<init> ()V (2)
4:    return

void footest() <(Unknown attribute Example:
                        48 65 6c 6c 6f 20 77 6f 72 6c 64 21)>
Code(max_stack = 3, max_locals = 1, code_length = 7)
0:    iconst_2
1:    newarray                <int>
3:    iconst_0
4:    iconst_1
5:    iastore
6:    return
```

3 The Array Bounds Check Annotation Example

In this section, we will use the array bounds check attribute to illustrate the process of creating a new code attribute.

The classes in this example are located in the `soot.jimple.toolkits.annotation.arraycheck` and `.nullcheck` packages.

Clearly we must be able to repesent whether or not an array reference is safe. To do this, we first created the `ArrayCheckTag` class implementing (a subclass of) `Tag`. It is not an `Attribute` because the information is not in a form suitable for adding to a classfile and setting the information directly is meaningless. This class has a constructor with boolean parameters representing upper and lower bounds checks. If a parameter is `true`, the respective bound check is needed. The `getValue()` method converts the boolean values to a byte value where the lowest two bits represent the bounds checks.

```
/**
 * This tag represents the two bounds checks of an array reference.
 * The value <code>true</code> indicates that a check is xneeded.
 */
public ArrayCheckTag(boolean lower, boolean upper)
{
    lowerCheck = lower;
    upperCheck = upper;
}

/** Returns the value of this tag as a one-byte array for inclusion in
 * the classfile. */
public byte[] getValue()
{
    byte[] value = new byte[1];

    value[0] = 0;

    if (lowerCheck)
        value[0] |= 0x01;

    if (upperCheck)
        value[0] |= 0x02;

    return value;
}
```

We designed an algorithm to analyze array bounds checks. The final phase of this algorithm attaches the analysis results to the various units as tags. This is accomplished with the following code:

```
Tag checkTag = new ArrayCheckTag(lowercheck, uppercheck);
stmt.addTag(checkTag);
```

As previously explained, code tags are attached to units, but units themselves do not have attributes. Thus, an aggregator is needed to group the attributes. Now, a null pointer check elimination algorithm has already executed, attaching `NullCheckTag`s to units. An `ArrayNullTagAggregator` will collect the `NullCheckTag`s and `ArrayCheckTag`s, combining these two tags into a single `ArrayNullCheckTag` per method body.

```
public class ArrayNullCheckTag implements OneByteCodeTag
{
    private final static String NAME = "ArrayNullCheckTag";
```

```
    public String getName();
    public byte[] getValue();
    public byte accumulate(byte other);
}
```

The `ArrayNullTagAggregator` implements the `TagAggregator` interface. It is called while Baf is generating its backend code. The `refresh()` method clears accumulated tags; it is called when a new method is examined. The `aggregateTag` method accumulates one (unit, tag) pair, typically encountered during Baf's traversal of the units. Finally, the `produceAggregateTag` generates a fresh `CodeAttribute` with the name and (unit, tag) pairs we have collected from the method under consideration.

```
public class ArrayNullTagAggregator implements TagAggregator
{
    private boolean status = false;
    private List tags = new LinkedList();
    private List units = new LinkedList();

    private Unit lastUnit = null;
    private ArrayNullCheckTag lastTag = null;

    public ArrayNullTagAggregator(boolean active)
    {
        this.status = active;
    }

    public boolean isActive()
    {
        return this.status;
    }

    public void refresh()
    {
        tags.clear();
        units.clear();
        lastUnit = null;
        lastTag = null;
    }

    public void aggregateTag(Tag t, Unit u)
    {
        if(t instanceof OneByteCodeTag)
        {
            if (lastUnit == u)
            {
                byte[] v = ((OneByteCodeTag)t).getValue();
                    lastTag.accumulate(v[0]);
            }
            else
            {
                units.add(u);
                lastUnit = u;

                byte[] v = ((Attribute)t).getValue();
                lastTag = new ArrayNullCheckTag(v[0]);
```

```
                tags.add(lastTag);
            }
        }
    }

    public Tag produceAggregateTag()
    {
        if(units.size() == 0)
            return null;
        else
            return new CodeAttribute("ArrayNullCheckAttribute",
                                     new LinkedList(units),
                                     new LinkedList(tags));
    }
}
```

We examine the annotation process on a simple example, `foo.class`.

```
public class foo
{
    void footest()
    {
        int[] c = new int[2];
        c[0] = 1;
    }
}
```

After compilation with `javac`, we can use the `JavaClass` tool to inspect the contents of the class file.

```
public class foo extends java.lang.Object
filename                foo
compiled from           foo.java
compiler version        45.3
access flags            33
constant pool           14 entries
ACC_SUPER flag          true

Attribute(s):
        SourceFile(foo.java)

2 methods:
        public void <init>()
        void footest()

public void <init>()
Code(max_stack = 1, max_locals = 1, code_length = 5)
0:    aload_0
1:    invokespecial        java.lang.Object.<init> ()V (3)
4:    return

Attribute(s) = LineNumber(0, 1)

void footest()
Code(max_stack = 3, max_locals = 2, code_length = 9)
```

```
0:    iconst_2
1:    newarray               <int>
3:    astore_1
4:    aload_1
5:    iconst_0
6:    iconst_1
7:    iastore
8:    return

Attribute(s) = LineNumber(0, 5), LineNumber(4, 7), LineNumber(8, 3)
```

Great. Next, we annotate the class by executing

```
[plam@kerala soot]$ java soot.Main -A both foo
```

and inspect the annotated class file.

```
public class foo extends java.lang.Object
filename                foo
compiled from           foo.jasmin
compiler version        45.3
access flags            33
constant pool           14 entries
ACC_SUPER flag          true

Attribute(s):
        SourceFile(foo.jasmin)

2 methods:
        public void <init>()
        void footest()

public void <init>()
Code(max_stack = 1, max_locals = 1, code_length = 5)
0:    aload_0
1:    invokespecial        java.lang.Object.<init> ()V (2)
4:    return

Attribute(s) = (Unknown attribute ArrayNullCheckAttribute: 00 01 00)

void footest()
Code(max_stack = 3, max_locals = 1, code_length = 7)
0:    iconst_2
1:    newarray               <int>
3:    iconst_0
4:    iconst_1
5:    iastore
6:    return

Attribute(s) = (Unknown attribute ArrayNullCheckAttribute: 00 05 00)
```

We can see that an `ArrayNullCheckAttribute` has been added to the class file, and we can read the attribute data in hexadecimal.

Known shortcomings

Soot cannot currently preserve existing attributes in a class file when transforming and annotating it. In the `foo` example, any debug information from `javac` would be lost after annotation.

History

- October 6, 2000 : Initial version.