Soot Tutorial

What is Soot

Soot is a compiler infrastructure for Java applications that was started at the Sable Research Group at McGill University. It provides four intermediate representations for analyzing and transforming Java bytecode:

- Baf: a streamlined representation of bytecode which is simple to manipulate.
- Jimple: a typed 3-address intermediate representation suitable for optimization.
- Shimple: an SSA variation of Jimple.
- Grimp: an aggregated version of Jimple suitable for decompilation and code inspection.

One of the main benefits of Soot is that these four different Intermediate Representations (IR) can be useful for different kind of analyses. In our tutorial we are going to focus on using Jimple.

Jimple

The Jimple representation is a typed, 3-address, statement based intermediate representation. Usually expressions occur in just one assignment: `x = a + b` and branch statements with one expression `if a < b goto L1`. The Jimple IR comes in two flavors: internal and external Jimple. External Jimple IR is an ASCII text file whereas the internal Jimple IR is provided in the form of a Jimple API (Java classes). We are going to use this API for our analyses. The main characteristics of Jimple are (we have to take this into account for our analysis):

- The Code is stackless
- Expressions restricted to the least operands (normally 2) and these operands must be either constants or locals
- All locals must be explicitly declared and typed
- Some locals can be assigned special roles
- Only 15 kinds of statements, bytecode has over 200

Java Code	**Jimple Code**

`public int stepPoly(int x)`
`{`
`  if(x < 0)`
`  {`
`  System.out.println("error");`
`  return -1;`
`  }`
`  else if(x <= 5)`
`  return x * x;`
`  else`
`  return x * 5 + 16;`
`}`
	`public int stepPoly(int)`
	`{`
	`  java.io.PrintStream r1;`
	`  Example r0;`
	`  int i0;`
	`  r0 := @this;`
	`  i0 := @parameter0;`
	`  if i0 >= 0 goto label0;`
	`  r1 = java.lang.System.out;`
	`  r1.println("error");`
	`  return -1;`
	`  label0:`

	`if i0 > 5 goto label1;`
	`i0 = i0 * i0;`
	`return i0;`
	`label1:`
	`i0 = i0 * 5;`
	`i0 = i0 + 16;`
	`return i0;`
	`}`

Figure 1.

Main Data Structures in the Jimple API

Figure 2 shows the class hierarchy for the main data structures from the Jimple API

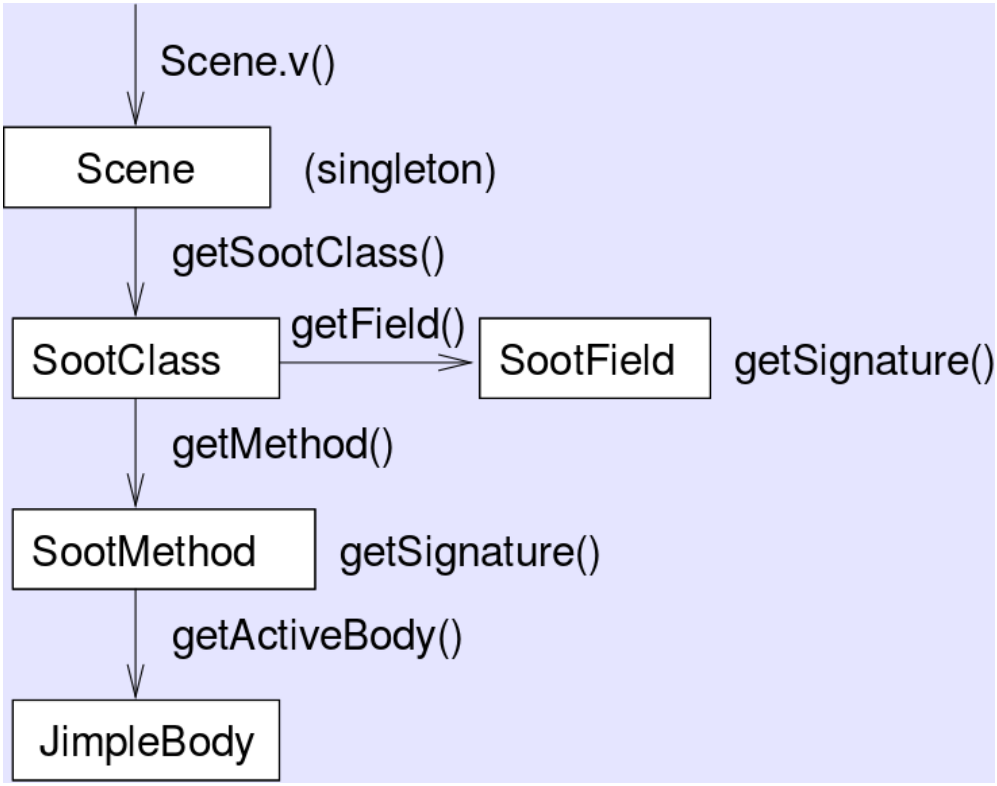

Figure 2

Scene. The Scene class represents the complete environment the analysis takes place in. Contains points-to information and call graphs.
SootClass. Represents a single class loaded into Soot or created using Soot.
SootMethod. Represents a single method of a class.
SootField. Represents a member field of a class.
Body. Represents a method body and comes in different flavors, corresponding to different IRs (e.g., JimpleBody).
Locals. Locals declared and used in a method. Locals are declared at the top of the method.
Show figure 1 with the following section highlighted

```
java.io.PrintStream r1;
Example r0;
int i0;
```

Units. A body is a chain of units. Each line in figure 1 is a unit. The Jimple API uses Stmt implementation of Unit.

```
i0 = i0 * 5;  =====> AssignStmt
```

Figure 3 shows the hierarchy between SootMethod, Body, Locals and Units. Bodies are composed by a chain (sequence) of locals and a chain of units (statements). Bodies can also have traps which represent control flow to catched exceptions.

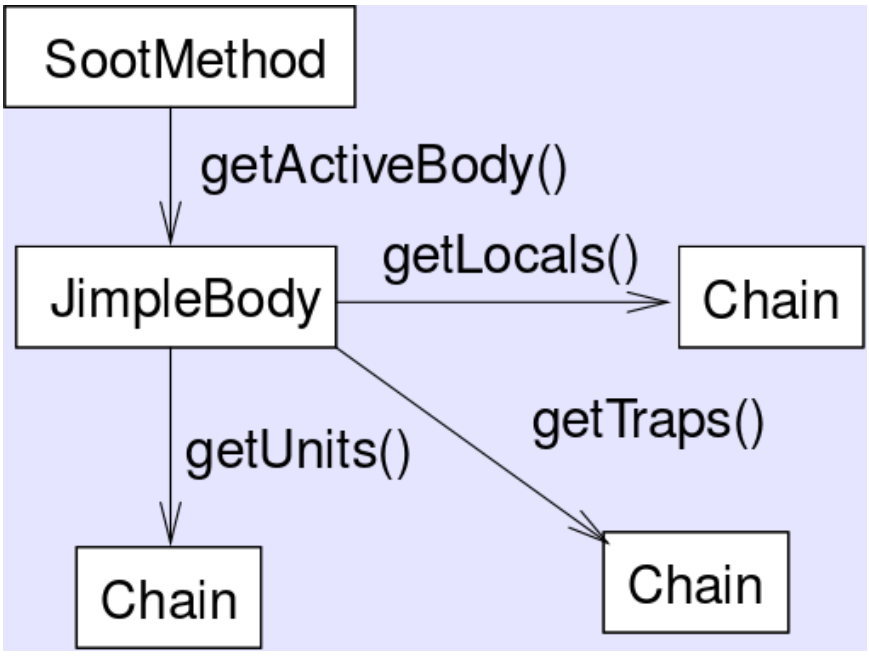

Figure 3

Soot Phases

As it was mentioned in class, program analysis framework and compilers are separated in several phases. In Soot these phases are called *packs*. Figure 4 shows the execution flow through packs in Soot. The naming convention is: the first letter defines the IR used (e.g. j for Jimple, s for Shimple); the second letter designates the role of the pack (e.g. b for body creation, t for user-defined transformations, o for optimizations); and the third letter stands for “pack”. These phases are used for intra-procedural analysis. The following section shows the phases use for inter-procedural analysis.

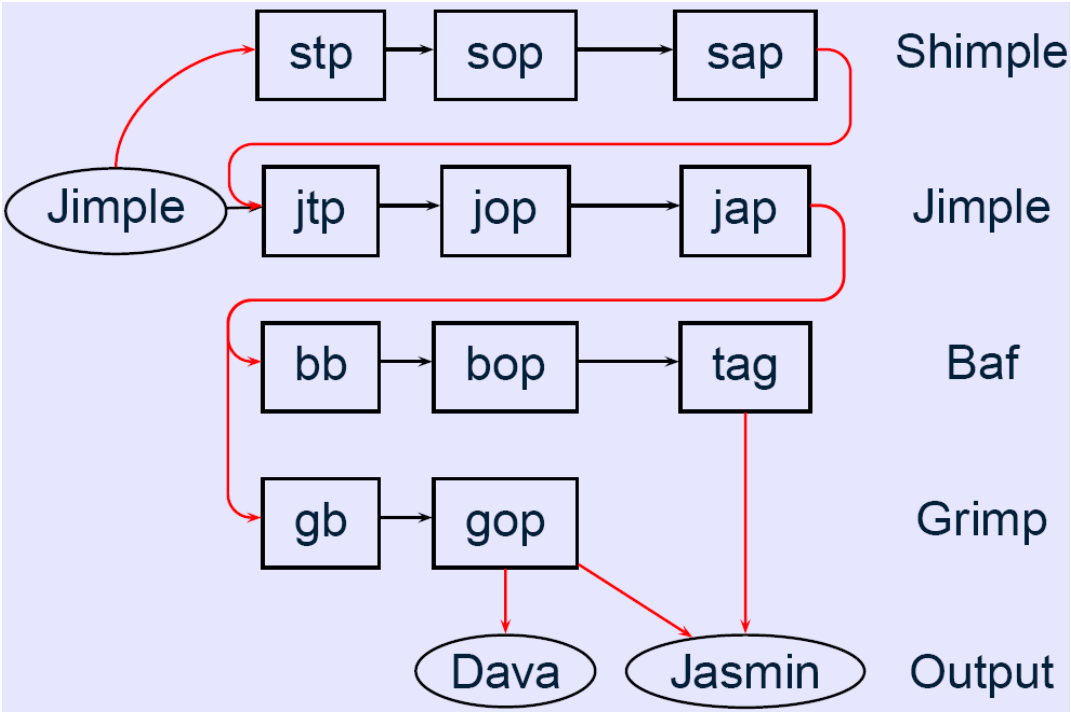

Figure 4

Inter-procedural Analysis

For inter-procedural analyses, we need Soot to work in *Whole-program mode*. For that we can use the flag *-w* to Soot. Similarly that for intra-procedural analysis, Soot is composed for several phases for inter-procedural analysis: cg (call-graph generation), wjtp (whole Jimple transformation pack) and wjap (whole Jimple annotation pack). For whole program optimizations we add the flag *-W* which adds the phase wjop (whole Jimple optimization pack). Figure 5 shows the execution flow for inter-procedural analyses.

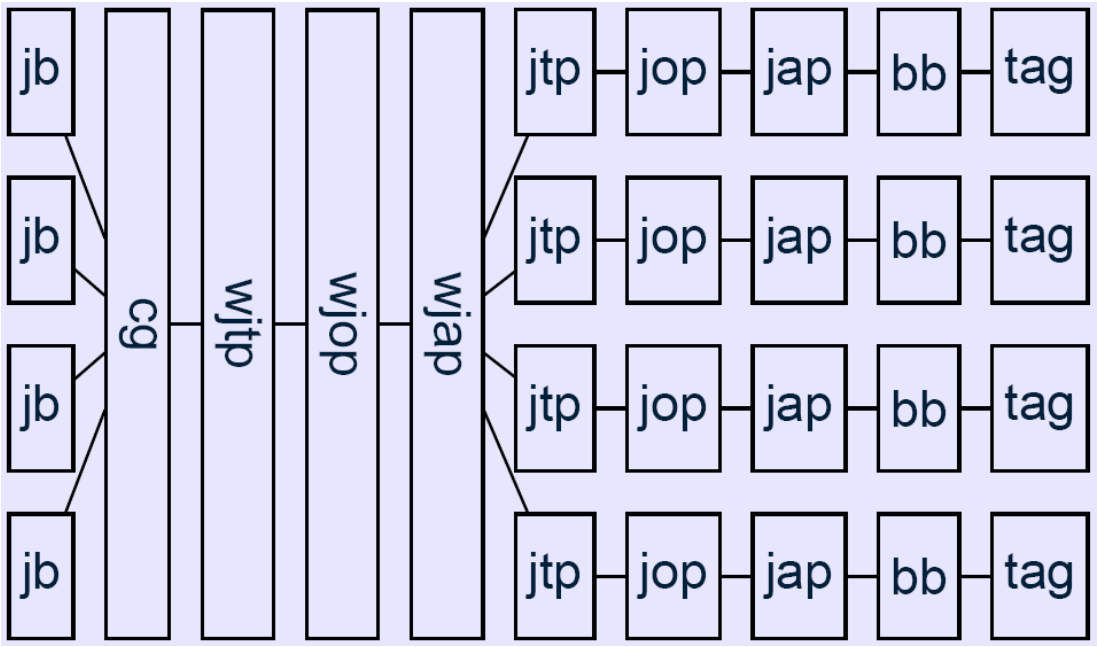

Figure 5

Setting Up Soot

Generate Jar file. We are not going to use the integration with Eclipse. We are going to use Soot as a library in our project that we will leverage for our analyses. I have included a script (deploysoot.sh) to download the project and build a Jar file for Soot in case you want to generate the last development version. Once we have this library, we can add it to the classpath of our project and start developing our analysis.

Steps to work on the tutorial

1. Install Java JDK 7 (1.7). For Windows and Mac you can download the distribution from their webpage
   http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
   For ubuntu:
   ```
   sudo apt-get install openjdk-7-jdk
   ```
2. Donwload Eclipse IDE from: http://eclipse.org/downloads/
3. Download Graphviz dot viewer:
   a. Windows: http://www.graphviz.org/Download_windows.php
   b. Linux: http://www.graphviz.org/Download_linux_ubuntu.php
   c. Mac: http://www.graphviz.org/Download_macos.php
4. Jd decompiler: http://jd.benow.ca/
5. Download git from: http://git-scm.com/downloads
6. Start eclipse and select a workspace directory

7. Clone the git repository for the tutorial in the workspace of Eclipse using the git command prompt in Windows and the terminal in Mac and Linux:
   ```
   cd /path/to/workspace
   git clone https://github.com/danilo04/soottutorial.git
   git checkout coms641
   ```
8. Import a project in: *File* → *Import* → *Existing Projecs into Workspace* → *Browse* and select the directory of the project (*soottutorial)*.
9. Right click on the project (soottutorial) in the *Package Explorer* and click properties. Click en *Java Build Path* and then click in Add JARs… Find soot-develop.jar in the *libs* directory
10. From this point, we will start the tutorial

References

1. Laurie Hendren, Patrick Lam, Jennifer Lhotak, Ondrej Lhotak, , and Feng Qian. Soot, a tool for analyzing and transforming java bytecode.
2. Einarsson, Arni, and Janus Dam Nielsen. "A survivor's guide to Java program analysis with soot." *BRICS, Department of Computer Science, University of Aarhus, Denmark* (2008).
3. https://github.com/Sable/soot/wiki/Fundamental-Soot-objects