Soot Command-Line Documentation

Patrice Pominville
patrice@sable.mcgill.ca

March 8, 2000

1 SYNOPSIS

Soot can be invoked in the following ways:

```
soot [option]* [classname]+
soot --app [option]* classname
```

2 DESCRIPTION

This manual page documents the command line version of the `soot` bytecode compiler/optimizer tool.

When given one or several *classname*s that refers to a Java type, and no `--app` option, `soot` will attempt to resolve it by finding a file containing the given type. Once `soot` has located such a file, it will read in its contents, perform transformations on its bytecode and output the type in a specified output format. This mode of operation is referred to as running `soot` in *single-file mode*. All types specified on the commandline are resolved and processed. In this mode, the last file specified on the command-line serves as the main class, when such a notion is needed.

The `--app` argument can be used to activate `soot`'s *application mode*. In *application mode* `soot` will perform a transitive closure on the types listed in the constant pool of the type provided on the command line. `soot` will then proceed to transform the types in this closure. The closure will contain Java library types, as well as types particular to the application. By default, only the application-specific types will be processed by `soot`. This can be overridden by command line options. Clearly, in this case, the file specified on the command-line is the main class.

To resolve a type, `soot` uses the same semantics as the `java` command; `soot` looks for files containing types in the directories specified by the `soot.class.path` system property. This property serves the same purpose as `java`'s `CLASSPATH` environment variable. There is also a command line option to override the `soot.class.path` property. If there is no Soot classpath, then the external Java `CLASSPATH` is used. (There is a note for Windows users in the section describing the soot-class-path entry).

Once a type has been resolved and read into `soot`, various transformations can be applied to its code. These are described in the optimization section of the options below.

The Soot framework has 3 different internal representations: *Baf*, *Jimple* and *Grimp*. `soot` allows one to output a processed class either as a standard classfile or in the textual format corresponding to one of the above internal representations. Thus a class can be outputted as a `.baf` file, a `.jimple` file or a `.grimple` file that will contain textual representations for the *Baf* , *Jimple*

and *Grimp* internal representations respectively. Additionally a processed type can be outputed in the *Jasmin* assembler format, as a `.jasmin` file.

3 OPTIONS

3.1 Input Options

At present, only the textual representation of Soot's *Jimple* internal representation can be parsed. Soot can also read `.class` files directly.

--src-prec jimple By default `soot` will resolve types from `.class` files. If a type cannot be resolved from a classfile, `soot` will attempt to resolve it from a `.jimple` file. This option specifies the opposite policy: types are to resolved from `.jimple` files and only if this fails will an attempt be made to resolve them from `.class` files.

3.2 Output Options

-b, --b Produces `.b` files. These contain an abbreviated textual form for soot's *Baf* internal representation. It is easier to read than its non-abbreviated counterpart, but can also contain ambiguities; for instance, method signatures are not uniquely determined.

-B, --baf Produces `.baf` files that contain a textual representation for types in soot's *Baf* internal representation.

-j, --jimp Produces `.jimp` files. These contain an abbreviated textual form for soot's *Jimple* internal representation. Can contain ambiguous references.

-J, --jimple Produces `.jimple` files that contain a textual representation for types in soot's *Jimple* internal representation.

-g, --grimp Produces `.grimp` files. These contain an abbreviated textual form for soot's *Grimp* internal representation. Can contain ambiguous references.

-G, --grimple Produces `.grimple` files that contain a textual representation for types in soot's *Grimp* internal representation.

-s, --jasmin Produces `.jasmin` files for types. These can be understood by the jasmin bytecode assembler tool.

-c, --class Produces Java `.class` files executable under any Java Virtual Machine.

-d PATH Specifies that the outputted files are to be stored in PATH. This may be relative to the working directory.

3.3 *Application Mode* Options

These are only valid if `soot` is invoked in *application mode* by specifying the `--app` command line argument.

-x, --exclude PACKAGE Marks classfiles in `PACKAGE` (e.g. java.) as context classes. Jimple is not produced for context classes, but the `SootClass`, `SootField` and `SootMethod` signature objects are created.

-i, --include PACKAGE Marks the classfiles in `PACKAGE` (e.g. java.util.) as application classes. This option can be used to transform library types which by default are not transformed by `soot`.

-a, --analyze-context Relabels all context classes as library classes. This means that Jimple now gets produced for them.

--dynamic-path PATH Marks all class files in `PATH` as potentially dynamic classes. This allows aggressive optimization of applications for which the set of dynamic classes that can be loaded is known at compile time.

--dynamic-packages PACKAGES Marks all class files belonging to a package listed in PACKAGES (or one of its subpackages) as potentially dynamic classes. PACKAGES should be a list of packages separated by commas.
Example: `--dynamic-packages java.text.resources,spec.benchmarks._213_javac`

3.4 *Single File Mode* Options

--process-path PATH Process all classes in `PATH`. All the classes found in PATH will be loaded and transformed in single-file mode.

3.5 Optimization Options

-O --optimize Perform scalar optimizations on the classfiles.

-W --whole-optimize Perform whole program optimizations on the classfiles; this also enables -O.

3.6 Miscellaneous Options

--soot-classpath PATH Use `PATH` to resolve types. Overrides the `soot.class.path` system property.

Important note for Windows users Note that as of release 1, Soot will treat drive letters correctly, but under Windows the path separator *must* be a backslash (\), not a forward slash (/).

--final-rep baf When producing `.baf`, `.jasmin` or `.class` output, produce the output using Soot's *Baf* internal representation. See [VRGH+00] for details on this topic.

-t, --time Print out time statistics about transformations.

--subtract-gc Attempt to subtract garbage-collection time from the time stats.

-v, --verbose Verbose mode.

--debug Avoid catching exceptions and print debug information.

-p, --phase-option PHASE-NAME KEY:VALUE Set run-time option KEY to VALUE for PHASE-NAME (default for VALUE is true). This option is quite powerful; it is documented in phase options.

-A, --annotation ⟨ both — arraybounds — nullpointer ⟩] Enables both array bounds and null pointer annotation, or array bounds annotation only, or null pointer check annotation only, respectively.

4 EXAMPLES

- `soot --app --jimple -O -soot-class-path classes:/localhome/plam/JDKlib MyApp`

  Invokes soot in *application mode* for the type `MyApp`. All types to be resolved should be found in the working directory's `classes` subdirectory or `/localhome/plam/JDKlib`. Types will be resolved from `.class` files; if this fails, `.jimple` files will be considered. Only the application's classes will be processed; scalar optimizations will be applied to these. The transformed types will be outputted in *Jimple*'s textual representation as `.jimple` files.

- `soot --final-rep baf --src-prec jimple MyApp`

  Invokes soot in `single-file mode` for the type `MyApp`. Tries to resolve `MyApp` (and all other classes) from a `.jimple` file before looking for a `.class` file. Looks for the type in the directories specified by the `soot.class.path` system property. Outputs the type as a classfile, using the `Baf` internal representation to construct the bytecode.

5 BUGS

None.

6 History

March 8, 2000: Initial release.
March 11, 2000: Added note for Windows users.
September 1, 2000: Added the `--dynamic-packages` option.
October 6, 2000: Added the `-A`, `--annotation` option.

References

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible?. In *Compiler Construction, 9th International Conference, CC 2000*, April 2000.