

Soot support for whole-program analyses

Patrick Lam (plam@sable.mcgill.ca)

March 23, 2000

This note describes some features of Soot relevant to whole-program analyses. The developer should first understand both the `createClass` and `menagerie` documents.

1 Goals

In particular, the developer will understand:

- The role of the `Hierarchy` object, especially in relation to the `Scene`.
- Care and feeding of `InvokeGraphs`.

2 Soot Hierarchys

We have previously seen that the `Scene` is a container for all of the `SootClasses`. However, the `Scene` does not provide convenient access to the class hierarchy.

Soot provides the `Hierarchy` to answer inheritance-related questions. We start by instantiating a `Hierarchy` object with `new Hierarchy()`; it queries the `Scene` for all active `SootClass` objects and stores the data in instance variables.

Stale Hierarchies If a `Hierarchy` object is created, and the `Scene` is subsequently changed, by the addition or removal of a `SootClass`, the `Hierarchy` becomes stale, and attempts to call its methods will result in an exception.

The `Hierarchy` provides methods like `getSubclassesOf` and `isClassSuperclassOf`. In general, whenever the method name contains `Including`, the current class is included in the query; that is, the call `isClassSubclassOfIncluding(x, x)` returns true.

As of version 1.0.0, not all of the functionality on `Hierarchy` is provided.

3 Resolving Virtual Method Dispatches

In order to carry out certain program transformations, such as inlining, we must be able to query Soot about the list of possible targets for an `invoke` expression or statement. The `InvokeGraph` provides this functionality.

The usual way of getting an `InvokeGraph` is through the `ClassHierarchyAnalysis.newInvokeGraph()` method. This uses CHA to construct an `invoke` graph.

In the future, other ways of constructing `InvokeGraphs` might be provided, for instance with Variable Type Analysis. We do not currently supply a working VTA implementation with Soot; the initial implementation was developed for an old version and will eventually be updated.

The graph is inherently a bipartite one; one set of nodes is the set of callsites, and the other partition is the set of methods in the program. One can add sites to the `InvokeGraph`, and then add edges corresponding to invocations, from sites (previously added) to methods. This is done with, respectively, `InvokeGraph.addSite()` and `InvokeGraph.addTarget()`.

We can then query the `InvokeGraph` for a list of targets at any given callsite, using `getTargetsOf(Stmt site)`; to combine all callsites from a method, we use `getTargetsOf(SootMethod container)`. To get all of the methods reached from a given container, we use `getTransitiveTargetsOf(SootMethod container)`.

We can also query for a list of sites in any method, using `getSitesOf()`.

History

- March 23, 2000: Initial version.