Soot-skeleton: Bootstrapping a Static Analyzer by Examples

Abstract

As a frequently used control-flow analysis framework, Soot supports various kinds of analysis, for instance, the call graph and the reaching definition. Those analyses are fundamental to the study of code optimization and program slicing. And that's why the Soot has been used so frequently in both the industry and the research area.

Since its flow analysis framework has been abstracted into the configuration and options, it assumes its users to be skilled at static analysis. However, for those who want to deploy Soot as part of their projects but with little knowledge about the static analysis algorithm, writing a Soot configuration that generates the output they're looking for would usually frustrate them. And they usually end up enumerating all possible configurations manually since they are not sure about which algorithm would produce the correct output. To tackle this issue, we propose an automated tool that can infer the correct configuration with an input-output example-based specification written by users so that this tool can generate a configuration to set up the Soot. This tool will enable users to get started with a working static analyzer prototype in a short time without much static analysis domain knowledge.

Background and Motivation

General Background

Soot is a Java optimization framework that provides transformation and analysis for Java code. It's a powerful tool that allows programmers to use it as a static analysis tool for Java/Android programs. However, its problems are also explicit. Firstly, because of its lack of abstraction of its options at a high-level, both inexperienced and seasoned users are prone to make errors when crafting the configuration file. Besides, it assumes its users possess knowledge of static analysis concepts, so that many of its APIs don't have a detailed explanation. And this requirement creates a long learning curve for the newcomers. For the next section, we will shed more light on this point.

A Concrete Problem

DemoClass.java is our analysis target for the demonstrating purpose of this project. For testing the correctness in the context of a syntactically complex Java file, we include two overload functions. And what follows is the code snippet for this target.

```java
public class DemoClass {
    public void overloadTester() {
        overload(1L);
        overload(1.0f);
    }
    public void overload(long i) {
        System.out.println("int");
    }
    public void overload(float i) {
        System.out.println("long");
    }
    public static void main(String[] args) {
        DemoClass dc = new DemoClass();
        dc.overloadTester();
    }
}
```

Here is an example of configuration code of Soot that will analyze the DemoClass.class, the byte code of DemoClass.java, in the test-resource folder and retrieve its call-graph:

```java
      soot.G.reset();
      Options.v().set_process_dir(Collections.singletonList("test-resource"));
      Options.v().set_src_prec(Options.src_prec_class);
      Options.v().set_soot_classpath("test-resource");
      Options.v().set_whole_program(true);
//    Options.v().set_allow_phantom_refs(true);
      Options.v().set_ignore_resolution_errors(true);
      Options.v().set_no_bodies_for_excluded(true);
      Options.v().set_verbose(true);
      Scene.v().addBasicClass("DemoClass", SootClass.SIGNATURES);
      Scene.v().loadClassAndSupport("DemoClass");
      Scene.v().loadNecessaryClasses();
      SootClass testClass = Scene.v().getSootClass("DemoClass");
      PackManager.v().runPacks();
      CallGraph cg = Scene.v().getCallGraph();
```

For users who don't have much experience with static analysis with Soot API, this piece of code is hard to understand. We will explain some details about it below. This code is an example of Soot's configuration that generates the call graph of `DemoClass.java`. With little indicative information about the usage of those `Options`, users would found it's hard to craft a functional configuration file of Soot. And unfortunately, even if users can understand what this code is doing, the above code doesn't quite work. After running the corresponding code, it turns out to end up with following exception.

```
java.lang.RuntimeException: Phantom refs not allowed
        at soot.SootMethod.setPhantom(SootMethod.java:211)
        at soot.SootResolver.processResolveWorklist(SootResolver.java:158)
        at soot.SootResolver.resolveClass(SootResolver.java:134)
        at soot.Scene.loadClass(Scene.java:942)
        at soot.Scene.loadClassAndSupport(Scene.java:927)
```

This error is telling the user that his code doesn't allow the phantom reference option. To understand the actual problem here, users will have to understand what phantom reference is. Phantom references are most often used for scheduling pre-mortem cleanup actions. Unlike any other reference, phantom references are not automatically cleared by the garbage collector as they are enqueued. An object that is reachable via phantom references will remain so until all such references are cleared or themselves become unreachable. Let's come back to the problem itself. After decompiling this each class and debugging through the call stack, we found that this error is invoked because every time when the users try to load a new class and allow the whole-program mode in their configuration, Soot will firstly set each processing class as a phantom class and then set each method as a phantom method. If the configuration doesn't include phantom options, then it's going to generate the `RuntimeException` listed above.

In short, the main disadvantage is that Soot only tells users the symptom without explaining why the configuration should allow the phantom reference, not to say how to fix it automatically. Even though that reason seems relatively simple, it demands the users to possess sufficient knowledge of phantom reference and to spend more time debugging. For those who are not proficient static analysis researchers, encountering this error would be unpleasant. Therefore, we propose to build an automatic inference tool that treats Soot as a semi-blackbox and try to find the correct configuration by enumeration.

Approach

Architecture

Current soot-skeleton architecture can be split into two separate modes: generator and runner. To begin with, we build the generator that can take in users' examples in uniform syntax. Our tool is currently supporting the example to be either a fragment of the call graph or a fragment of the reaching definition. After digesting this information, the generator will infer the configuration set-up based on the examples that users provide. Then the generator will generate the best fitting configuration that fits in users' examples as the output. After that, users can take a look at both the generated configuration and the corresponding output. Take the call graph analysis as an example. If the user finds the call graph output is missing some edges that he demands, then he is capable of including more example fragments to refine the output configuration and iterate through this process several times. And his configuration will gradually be refined during this process.

Besides, soot-skeleton also provides the runner module. The functionality of this module is based on the output of the generator. If users finish finding the configuration by running the generator, then they can run on a different input with the runner using the inferential configuration.

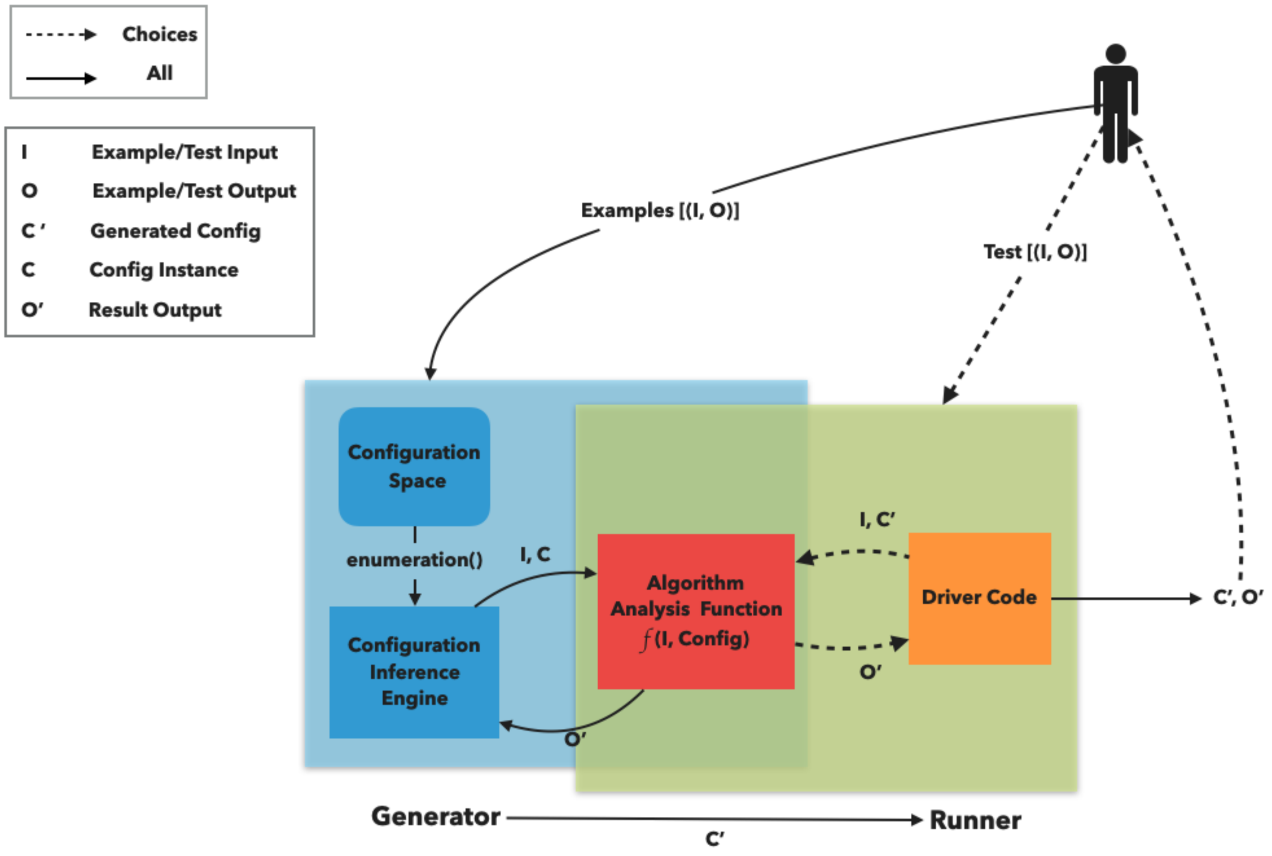

Figure 1: Architecture Diagram

Algorithm

The soot-skeleton validates and infers the users' input. Our tool supports two frequently used static analysis algorithms: the call graph and reaching definition. For each different kind of input, the user could anticipate that soot-skeleton's core analyzer to take different approaches to find the configuration. To begin with, soot-skeleton initially validates that the user data is formatted and store it locally. From there, it will invoke the core configuration inference engine to determine the best-fit algorithm.

Next, we will shed more light on the inference algorithm we built. We used enumerative synthesis as our main inference approach. The enumeration logic is indicated in the above graph. It works as the inference search engine persistently enumerates the Soot configuration C from the Soot configuration space, dispatches C and user input I to the algorithm analysis function $f(I, config)$, then retrieves the output O'. It will keep doing so until either the output O' passes the validation test or all combinations of configuration have been used up. Additionally, for the purpose of demonstration, we will use a simplified binary options case to explain the enumeration ordering. Suppose that we have two configuration options U and V. Each option has False value and True value respectively. Note that a complete configuration C needs to have both U option and V option. Thus, the configuration space has 4 entities in total. Each blue square in Figure 2 represents a complete configuration C. In the enumeration case, the search engine will follow the order that is indicated by the arrow and evaluate each configuration separately. In this example specifically, it will start off with all configuration option that has been set to true, and eventually reach the configuration in which every option has been set to false.

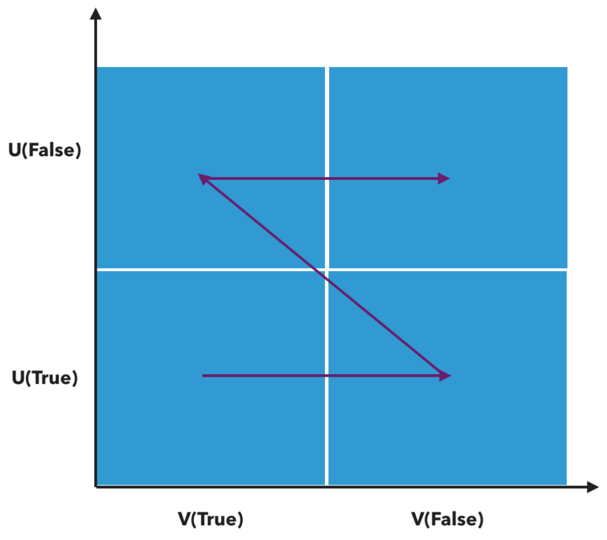

Figure 2: Enumeration Ordering

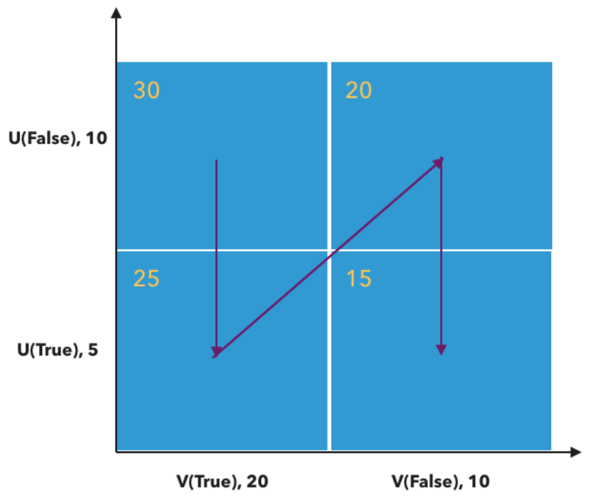

Figure3: Ranking Ordering

In addition to enumeration, we also attempt to use a ranking system for each option or the whole configuration vector inside the configuration space. For each configuration, the core analyzer of soot-skeleton contains a predefined function $f_p(C) \to X$ that would take one configuration instance as a input and output its according rank X. The rank for each configuration option was calculated by the times it has been utilized in a real configuration set-up. And the weight for each configuration as a whole is the sum of all the rank for each individual configuration option. More importantly, after the inference engine starts inferring, it begins with the configuration that has the highest weight. The reason it would start from this instance of configuration is, conceptually, it has the highest probability to be the best-fitting configuration because users in the real-world use it most frequently. Once the inference engine has done processing that configuration, it will proceed to the configuration with the second-highest rank and keep doing so iteratively until either the output passes the validation test or all combinations of configuration have been used up. Take the same binary entity in the previous enumeration case as an example. In Figure 3, we can see that each option was assigned a rank. For the demonstration purpose, the rank is assigned manually. Since we need to choose the 2 options U and V together to construct a complete configuration, their sum is the weight for its configuration in the 2-d dimension. And the inference engine will start with the U that has been set to False and V that has been set to True since this configuration has the highest rank when compared to any other configuration instance. After processing the first one, it will then apply the configuration with U that has been set to True and V that has been set to True since this configuration stands for the configuration with the second-highest rank. In this way, the ranked inference will follow the order that is indicated by the arrow in the graph. And that ordering is exactly the ranking ordering we constructed for soot-skeleton.

Implementation

Link to the Github: https://github.com/MarkGaox/soot-skeleton

This section will discuss further about the actual implementation of soot-skeleton. And we will discuss the implementation of the generator and the runner respectively.

Generator

To begin with, let's explain the essential logic of the generator. To run the generator, users need to provide an initial parameter and $Examples[(I, O)]$, input-output example pairs, which was indicated in the architecture Figure 1. And they need to be written in YAML format. After accepting this examples, soot-skeleton will store the example locally and invoke the configuration inference engine. And the engine will infer the correct configuration and generate it in the end. Then, let's explore how the generator was implemented.

In soot-skeleton, the generator takes the responsibility of inferring the configuration. To begin with, we implemented the interface to accept the input-output examples pairs and initial parameters. It will take in both of them in two YAML files and in a specific format. For the examples YAML file, users need to arrange the example in the format of input and its corresponding outputs which the user is requesting. The following is the corresponding format.

```
allClasses:
  <@target name>:
    "<@ input1>": ['<@ output1>', '<@ output2>', ...]
    "<@ input2>": ['<@ output1>', '<@ output2>', ...]
```

Take the call graph as an example, the input should be filled with the full signature of the caller method and the output should be specified by the callee's full method signature which represents the output call graph statement they're demanding. Note that the statement could be either the call graph statement or a reaching definition statement. And the generator will infer the type of example input. As for the initial parameter, it's essentially a setup for soot-skeleton. In this file, the users need to express the name of the example, the relative file path to that example, and the output path. The output path will later be used as the path for the generated configuration after soot-skeleton finishes inferring. Taking in all this information, the generator will parse and save them for the purpose of future comparison. Then, it will invoke the configuration search engine which was denoted as a blue square in Figure 1.

The configuration inference engine can be understood as a for loop that is written below. It will constantly enumerate configuration C from configuration space of Soot. As denoted as the central red square in the architecture diagram, the algorithm analysis function $f(I, Config)$ is implemented as the driver code of Soot framework. This function will take in the user input I, that is extracted from the user examples, and the enumerated configuration C, and, in the end, return the output O' that is produced by Soot framework under the context of C. Before spawning the final output, the validating function will take control. It will compare the O' with user example output O. If this O' covers everything that the user is demanding, it means this configuration instance is correct. Then our tool is going to generate the inferential configuration, namely C', in the format of YAML and in the given output path. If that's not the case, the inference engine will continue enumerating another configuration and test it in the same way until either one configuration instance passes the validation function or every configuration combination is used up.

```
for (C ← enumeration(Configuration Space)) {
   if (all(validate(f(I, Config), O) for (I, O ← examples)) {
        return C'
    }
}
return None
```

Runner

The runner part of this project provides a convenient approach for users to test their analysis target after they use the generator to infer the configuration. This module is denoted as the big green rectangle in the architecture diagram Figure 1. In order for it to complete the its operation, it takes in two things. Users need to provide a test input I and configuration C' that is generated by soot-skeleton previously. Accepting these information, the runner will dispatch the $I\ and\ C'$ into the algorithm analysis function f to retrieve the result O'. O', in this case, is the static analysis result produced by Soot framework under the context of configuration C'. And in the end, the runner will generate the O' to the users. By using the runner, users no longer need to write more configuration code in order to obtain the analysis result for the their test input. And they are able to do further analysis over other targets without writing more code, which reliefs their burden on doing static analysis.

Static Analysis Algorithms

For this section, we will present the static analysis algorithms we have used in soot-skeleton. Our tool incorporated two kinds of static analysis algorithms, the call graph analysis, and the reaching definition analysis. And we will explain them separately.

Call Graph

The call graph is the control flow graph that exists inside a program. It represents the calling relationship between procedures. The following graph is an example piece of a call graph. Each entity in this graph represents a procedure, and each edge presents the calling relationship between two procedures.

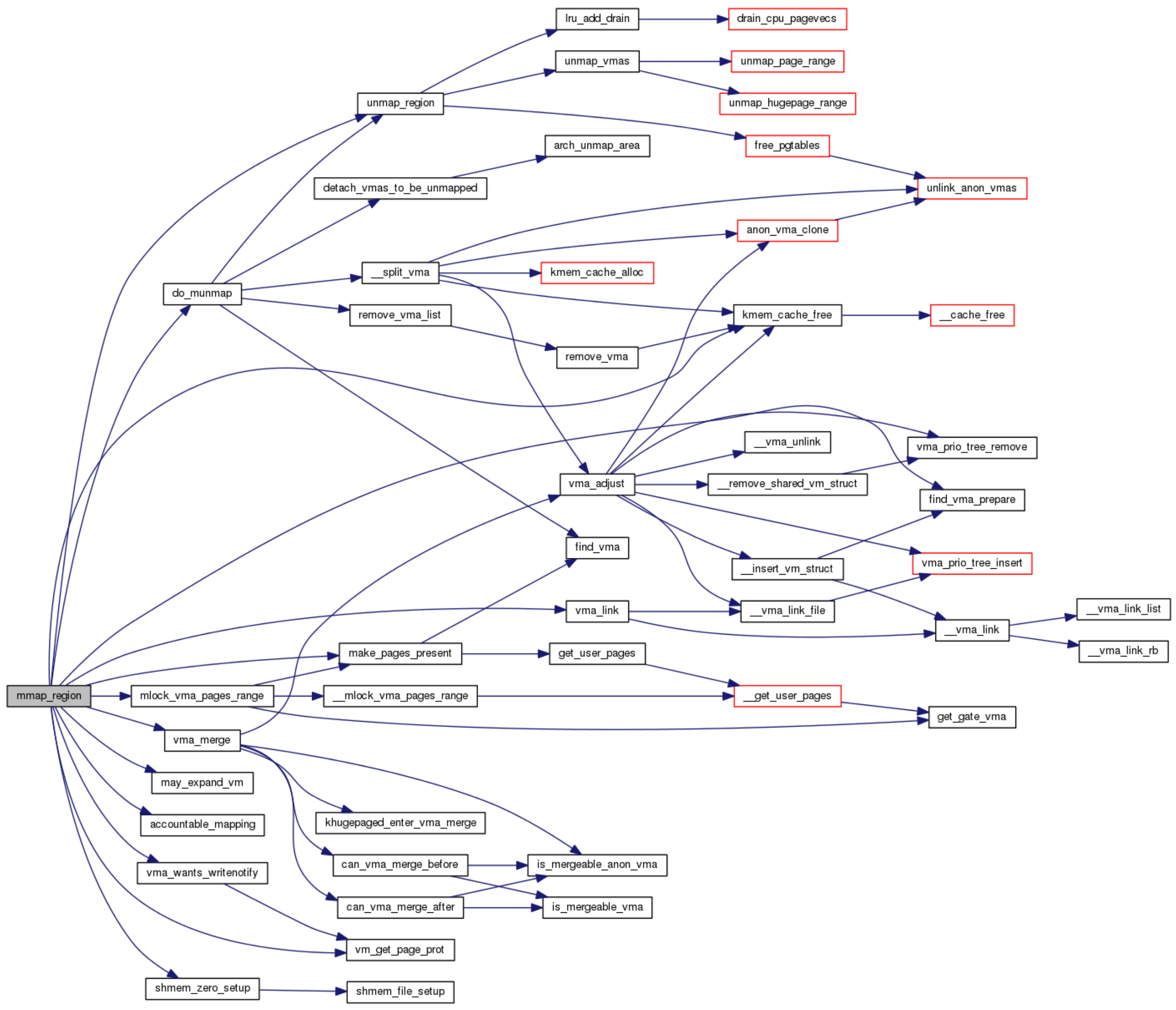

Figure 4: Call Graph Result Example

For the call graph analysis, our tool supports two frequently used algorithms of Soot, Class Hierarchy Analysis (CHA) and Soot Pointer Analysis Research Kit (SPARK). CHA is a simple version of call graph analysis since it assumes every variable can point to any object of that type in a program. This assumption used by CHA is sound, but, at the same time, might creates an imprecise call graph. Nevertheless, it's still valuable to the users since it can be used as a starting foundation for constructing a more precise analysis. In addition, we also incorporated the SPARK algorithm. SPARK is a more precise algorithm at the cost of speed when compared with CHA. It's a framework inside Soot for supporting point-to analysis in Java. It supports both subset-based and equivalence based points-to analysis and anything in between. And Soot uses this algorithm as the basis for the call graph construction.

Reaching Definition

Another static analysis algorithm soot-skeleton supports is the reaching definition analysis. Before explaining this algorithm, we would have to mention the inter-procedural, finite, distributive subset (IFDS) problems. IFDS problem is an inter-procedural analysis problem with a finite set of data flow facts and applicable distributive data-flow function over all the facts. And the reaching definition problem is one kind of IFDS problem. It's trying to determine what definition of the variable should be available at a given point of a program from the hint of the earlier program. The following graph is a simplified example of reaching definition. In this graph, “In” represents the entry for the program, and “Out” indicates the end of execution. From this graph, we can see that the code has three points, d1, d2, d3 respectively. The definition of y at d1 is 3. Before the code continues executing to the point of d2, this definition is still reachable. However, after d2 is executed, the definition at d1 is no longer valid since the variable y has a new reachable definition of 4. When the code continues executing until the point of d3, d1 is still no longer reachable. Therefore d2’s definition of y is the actual reaching definition at the point of d3. This example is intensively simplified since in the real world we also need to take care of the control flow when doing a flow-sensitive data flow analysis.

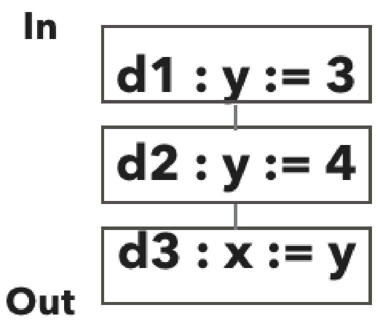

Figure 5: Reaching Definition Example

To support this analysis in soot-skeleton, we implemented our own reaching definition transformers to transform the code into the set of reaching definitions. Besides, we used the IFDS solver that was implemented in Hero, an extended project of Soot. With these two things, we used the same API of Soot in the call graph analysis to analyze the reaching definition.

Case Study

This section will use the example of DemoClass.java that was introduced at the beginning to demonstrate how soot-skeleton facilitates Soot users with their working efficiency. Assume a user would like to analyze the call graph for a big Java class. With little knowledge about the static analysis, he is unable to write the correct Soot configuration code since he doesn’t know which static analysis algorithm would generate the call graph he’d like to acquire. Fortunately, he is sure about what a part of the full call graph for that application would look like. Therefore, he would like to use the soot-skeleton to find the Soot configuration that fits his needs. As described in the previous section, users need to provide examples of the input and output they are looking for. Suppose that it’s named examples.yaml. And here is a snippet of the examples.yaml.

```yaml
DemoClass:
  "<DemoClass: void overloadTester()>":
                                      ['<DemoClass: void overload(float)>',
                                       '<DemoClass: void overload(long)>']
  "<DemoClass: void main(java.lang.String[])>":
```

```
                                         ["<DemoClass: void <init>()>",
                                         "<DemoClass: void overloadTester()>"]
```

This piece reveals several things. It indicates the name of the target should be “DemoClass” and the target should has methods `overloadTester()` and `main()`. The statements that are included in the square bracket are corresponding call graph relationships that the user intends to acquire in the result output. After compiling the soot-skeleton, the user could run in the generator mode with following command-line instruction:

```
java -jar target/soot-skeleton-1.0-SNAPSHOT-jar-with-dependencies.jar -cfg config.y
aml -exp examples.yaml
```

The command “-cfg” indicates the option for the initial parameter. The initial parameter, as mentioned before, includes the path for analysis target and generated configuration. The command “-exp” indicates the path for the example that the user wants to provide. After running this command, soot-skeleton would take in the input and enumerate through the configuration space. Eventually, soot-skeleton will output the call graph and the synthesized configuration result, which are listed below.

Call Graph Output:

```
<DemoClass: void overloadTester()> : <DemoClass: void overload(float)>
<DemoClass: void overloadTester()> : <DemoClass: void overload(long)>
<DemoClass: void overload(float)> : <java.lang.System: void <clinit>()>
<DemoClass: void overload(float)> : <java.lang.Object: void <clinit>()>
<DemoClass: void main(java.lang.String[])> : <DemoClass: void overloadTester()>
<DemoClass: void main(java.lang.String[])> : <DemoClass: void <init>()>
<DemoClass: void overload(long)> : <java.lang.System: void <clinit>()>
<DemoClass: void overload(long)> : <java.lang.†Object: void <clinit>()>
<DemoClass: void <init>()> : <java.lang.Object: void <init>()>
```

Configuration Result:

```
CG_Safe_New_Instance : true
CG_Spark_Verbose : true
CG_Spark_OnFlyCg : true
IGNORE_RESOLUTION : true
VERBOSE : true
NOBODY_EXCLUDED : true
SET_APP : true
CG_Cha_Enabled : true
WHOLE_PROGRAM : true
ALLOW_PHANTOM_REF : true
CG_Spark_Enabled : true
```

The call graph output is all the relationship that was found under the context of the generated configuration. And we can see that it contains every example input-output pair the user provides as examples along with some other call graph relationships. It indicates that this output satisfies users' requirements. Further, getting such configuration, users can use this configuration to test his non-deterministic big application in the runner mode of soot-skeleton. If he knows that some other call graph edges are not included in the result, he could then incorporate these relationships into the examples.yaml and run the generator again to gain a more accurate configuration. Note that for this example the output configuration is all marked as true but that’s not always the case.

Related Work

The primary goal of soot-skeleton is to automate the process of static analyses by input-output examples. The large body of soot-skeleton is based on the work of Soot. We used its data-flow analysis framework to implement this automation tool. The primary difference between our tool and Soot is that we used the example-driven approach to optimize the user experience of using Soot. With further notice, our example-driven approach has been widely adopted. Microsoft Excel used this approach to automatically synthesis the programming on the spreadsheet by letting the users provide the input-output. Perelman used the input-output strategy to enable end-users to create a program in a domain-specific language. One similarity sharing between these works is the programmers are set free from writing more code by providing input-output examples which they intend to accomplish.

In a similar approach, Kellogg has developed a tool, sassy, to synthesis the static analysis by examples, which shares a largely similar perspective with this project. We both used the example-driven approach to help the user quickly adapted themselves to the static analysis. The primary difference is that we define the domain of the problem (call graph and reaching definition) while his work lets the user define the domain in order to synthesize the analyses. We believe that in terms of entry-level researchers, soot-skeleton poses lower barriers to them to do static analysis and that is the essential motivation of soot-skeleton. Furthermore, since our static analysis is based on Soot ‘s built-in flow-analysis framework, our main interest is the configuration inference while the sassy let the user provide the configuration.

Our tool relies on the configuration inference. We utilized two approaches to facilitate our inference as we described in the implementation section: enumeration synthesis, and ranking synthesis. As far as we know, soot-skeleton is not the first practice to implement configuration generation with enumerative searching. Hutter and Hoos developed ParamILS which can search for the configuration of a parametric algorithm with the user given example algorithm. By letting the user provide the configuration space, ParamILS searches the whole configuration space and evaluates each combination, which is the same approach we implemented initially. On the other hand, the ranking approach we proposed is also applied in the optimization area. When accelerating the tensor program, researchers uses the similar rank-based objective to optimize the workload to further accelerate the tensor.

Future Work

Since our current implementation is limited to Java byte code, one of several objectives we intend to accomplish in the future is to expand our implementation with versatile input, for instance, the real world android application, because the baseline of this analysis is already supported by Soot. Furthermore, we also would like to implement the ranking system we proposed. This would require us to use a benchmark to find the optimal weights. Overall, we are convinced that soot-skeleton can facilitate new-comers of static analysis over Java to quickly adapt themselves to deploying static analysis into their projects. We are confident that our approaches can solve the actual problem we mention at the beginning of this paper.

Reference

https://mailman.cs.mcgill.ca/pipermail/soot-list/2019-April/009107.html

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *ACM SIGPLAN Notices*, volume 49, pages 408–418. ACM, 2014.

Hutter, Frank et al. “Automatic Algorithm Configuration Based on Local Search.” *AAAI* (2007).

Chen, Tianqi, et al. “Learning to Optimize Tensor Programs.” *ArXiv:1805.08166 [Cs, Stat]*, Jan. 2019. *arXiv.org*, http://arxiv.org/abs/1805.08166.

Sable/soot

Soot - A Java optimization framework. Contribute to Sable/soot development by creating an account on GitHub.

github.com

Sable/soot • github.com

Sable/heros

IFDS/IDE Solver for Soot. Contribute to Sable/heros development by creating an account on GitHub.

github.com

Sable/heros • github.com

kelloggm/sassy

Static AnalySis SYnthesis. Contribute to kelloggm/sassy development by creating an account on GitHub.

github.com

kelloggm/sassy • github.com