Artifact : Directed Test Generation to Detect Loop Inefficiencies

Monika Dhok
Indian Institute of Science, Bangalore
monika.dhok@csa.iisc.ernet.in

Murali Krishna Ramanathan
Indian Institute of Science, Bangalore
muralikrishna@csa.iisc.ernet.in

1. SCORECARDS

1.1 Insightful

Performance is important for software applications. Performance issues are found in well tested commercial products, since they are hard to detect. Many effective techniques are proposed to detect performance bugs automatically [2, 1]. These techniques address variety of performance issues. Redundant loop traversal is one of the primary sources for performance issues in many Java libraries. Recently, static and dynamic analysis approaches are developed to detect these problems effectively. The dynamic analysis technique like [2] takes a set of tests as input and monitors execution to detect repetitive accesses. This approach is ineffective if used with unit tests since unit tests are targeted towards functional correctness. Therefore, manually written tests are provided as input which is time consuming and tedious.

Static analysis techniques [3] are also proposed to detect redundant loop traversals. However, programmer has to manually confirm the validity of such defects. We address the important problem of automatically generating tests to detect redundant loop traversals. These tests are helpful for dynamic analysis tools to effectively locate performance issues in the program without investing manual efforts. Such kind of approach is also helpful for static analysis, since it can be used to confirm the validity of bugs based on the execution time. Similarly, it can be used validate the bug fixes automatically.

1.2 Useful

Redundant traversal of the loop under nested conditions affect the performance significantly. In this work, we propose an automatic dynamic analysis technique to generate targetted tests that expose loop inefficiencies. Writing such test manually by analyzing hundreds of classes is impractical as it is time consuming and tedious. The proposed approach has detected a number of bugs across several Java libraries. Our reported bugs are confirmed and fixed by developers[1].

Our artifact is cost-effective. On an average, it takes around 10 minutes for end to end analysis of a class. This involves generating initial random tests, performing dynamic analysis on each test, building a callgraph on complete project and callgraph traversal. We evaluate our approach on 7 benchmarks which include `Apache collection`, `PDFBox`, `Groovy`, `Guava`, `JFreechart`, `Ant` and `Lucene`. The given artifact contains a script to execute all the benchmarks together and takes around 2 hours to run all the experiments.

[1] Our bug report https://sourceforge.net/p/jfreechart/bugs/1147/ has been acknowledged by developers and our suggested fix is incorporated.

1.3 Usable

The artifact is provided in the form of compressed tar.gz file. The uncompressed folder contains a VM image which is ready to use. The execution environment is set up in this image by installing all the required dependencies, compiling all the benchmarks and other configurations. The compressed tar.gz can be downloaded from `http://drona.csa.iisc.ac.in/~sss/tools/fse16`

The uncompressed folder also contains a README file which describes in detail how to use this artifact with snapshots at various stages of execution. This file provides detailed description of all the components present in the artifact. It describes how to analyze any given class using our approach. It also describes how to run all the benchmarks to validate data in the paper. We provide a high level script getData.sh in artifact folder on the desktop which runs all the benchmarks together. This script generates data for figure9 and figure10. The corresponding graphs are also generated for easy validation.

References

[1] K. Nguyen and G. Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 268–278, New York, NY, USA, 2013. ACM.

[2] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[3] O. Olivo, I. Dillig, and C. Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 369–378, New York, NY, USA, 2015. ACM.

2. ARTIFACT DETAILS

2.1 Introduction

The artifact is a 4GB tar.gz file, and is available at `http://drona.csa.iisc.ac.in/~sss/tools/fse16` The uncompressed folder consists of a VM image, and a README file which describes the instructions along with snapshots for clear understanding of the artifact. The virtual image provided has the required execution environment set up to run our analysis. Packages like maven, ant and gnuplot are preinstalled for building benchmarks and generating graphs. We have installed both Java7 and Java8 in the virtual machine. This is necessary since random test generators like Evosuite work with Java8 and soot works with Java7.

2.2 Hardware dependencies

1) Virtual box with version 5.0.18
`https://www.virtualbox.org/wiki/Download_Old_Builds_5_0`
2) System with at least 12GB of main memory.
2) System with at least 8 cores (preferably).

2.3 Installation

To execute virtual machine :
1) Download and uncompress the artifact (around 10GB).
2) Open VirtualBox.
3) Create a new VM by clicking machine –> new.
4) Give a name to the new VM.
5) Select Linux type and Ubuntu (64bit) version.
6) Click Next, select 4GB of RAM and select Next again.
7) In the hard drive selection screen, select the option to "use an existing hard drive file", then select artifact.vdi file containing the VM image that you just downloaded. Go to settings->system->processor and assign 4 cores to VM. Start the VM with `Username : artifact Password : 123`

2.4 Description

The tool is present in the folder named artifact on the desktop which is organized as:

- *versions* : original and modified versions of benchmarks.
- *randoop, evosuite* : contains scripts to generate and parse randomly generated tests.
- *sootAnalysis* : static and dynamic analysis.
- *perfTests* : tests generated to detect loop inefficiencies.
- *expected-perfTests* : expected generated tests to detect loop inefficiencies.
- *figures* : displays generated figures along the dimensions of figure9 and figure10 in the paper.
- *benchmark* : contains the program under analysis.
- *bugs* : a file with detailed description of the bugs detected.

The artifact folder contains a copy of the paper named paper.pdf which has minor changes over the submitted version. The updated version can be followed during evaluation.

2.5 Experimental setup

We describe three steps that might be followed as part of evaluation. We have used the environment variable `$fsehome` in the subsequent discussion which points to the artifact folder on the desktop. The first step shows the working on a simple example. This step describes the summary generated, and how it is leveraged further for synthesizing tests. In the second step, we show a demonstration on a class. The third step runs all the benchmarks together and generates data for Figure 9 and Figure 10 as shown in the paper. [You can skip STEP 1 and 2 to directly run benchmarks].

2.5.1 *STEP1 : Analyzing a simple example*

We have added examples in $fsehome/sootAnalysis/test folder so that one can get familiar with the tool.
1) To run these examples, `cd $fsehome/sootAnalysis`
2) Execute `./sampleTest.sh` ; outputs the summaries for each method in Type.java and the generated tests. [Run other examples using instructions at the header of sampleTest.sh]

2.5.2 *STEP2 : Analyzing a class from benchmarks*

Consider the class *CollectionBag* from collections which is placed in the package *org.apache.commons.collections4.bag*
1) Change the working directory to evosuite
`cd $fsehome/evosuite`
2) Remove contents of benchmark and perfTests
`rm -rf $fsehome/benchmark/*`
`rm -rf $fsehome/perfTests/*`
3) Copy class files of collections to benchmark
`cp -r $fsehome/versions/collections/bin/* $fsehome/benchmark/`
4) ./Puffer.sh -R <packageName> <className> for our analysis
For this example, the command is :
`./Puffer.sh -R org.apache.commons.collections4.bag CollectionBag.`
This step takes around 10 minutes depending upon the initial number of tests generated and the performance tests are stored in $fsehome/perfTests/ folder. It outputs 5 tests with different patterns for one redundant loop traversal. One of which fails during compilation because of invalid type (Section 4.1). This step may not generate any tests if the randomly generated input tests do not execute inefficient loop.

2.5.3 *STEP3*

Running all the benchmarks together to generate data for Figure 9 and Figure 10 :
1) Change the working directory to $fsehome. `cd $fsehome`
2) Execute `./getData.sh.`
This script performs the following operations.
i) Deletes pregenerated graphs for figure 9 and 10.
ii) Execute ./run.sh reuseSummary to analyse all the benchmarks using pregerated summaries
iii) Execute ./generateFigure9.sh to generate data for Figure9.
iv) Generate figure9 and figure10 in $fsehome/figures.
This script takes around 1.5 to 2 hours to finish execution. During execution of the scripts, you may observe some test failures. You can safely ignore them as those errors correspond to inconsistency of java versions.

The generated tests are stored in $fsehome/perfTests folder and methods with bugs are listed in the file TestsOutput.log. The generated figures (fig9.eps and fig10.eps) are saved in $fsehome/figures folder. Please note that, the results obtained will be roughly proportional to the performance ratios mentioned in the Figure9 and Figure10 of the paper. Figure 9 shows that the percentage performance improvement achieved using directed tests is more than random tests, i.e., grey bars should rise above black bars. Figure 10 shows that percentage performance improvement increases with size of the input collections.