Call Graph Construction for Java Libraries using OPAL

Michael Reif Michael Eichberg Ben Hermann Johannes Lerch Mira Mezini
Technische Universität Darmstadt
Darmstadt, Germany
{lastname}@cs.tu-darmstadt.de

As a companion to our paper, we provide the following artifacts:

- The list of dead methods found in the Oracle JDK7 update 80 (Windows) using the three discussed approaches (naïve, open- and closed package assumption).
- The source code of the implementation of the proposed algorithm.
- A docker container which contains the compiled version of the complete OPAL framework as well as the complete runtime to (re)compile and run OPAL.

1. LICENSE

The source code of the OPAL framework is licensed under the BSD 2-Clause license which is approved by the Open Source Initiative(OSI) [1]. OPAL is publicly available on Bitbucket: https://bitbucket.org/delors/opal and releases are made available on Maven Central: https://mvnrepository.com/artifact/de.opal-project.

2. INSIGHTFULNES

Given the general importance of libraries for the successful development of new software products, it is necessary to understand that static analyses that use classical call graph algorithms designed and implemented with applications in mind will generally produce unnecessary spurious results. Additionally, one needs to understand that it is impossible to design *the one call graph algorithm for libraries* that suits all needs. Given the provided artifacts – e.g. the lists of dead methods reported in the three cases – the effect of the different assumptions on the build call graph can directly be studied. Additionally, using the docker container it is always trivially possible to build the call graphs for further Java libraries under the respective assumptions. This makes it possible to value the relevance of the different assumptions w.r.t. building call graphs.

Furthermore, the list of dead methods also provides a first insight into quality issues of very long living software projects (> 20years) which are maintained by a huge and changing developer base.

3. USEFULNES

The list of dead-methods[2] can immediately be used by other researchers to compare other/new approaches against the proposed approach and, hence, will greatly facilitate evaluation of new research in the area of call graph construction for libraries and dead method identification. The source code can directly be used as the foundation for the implementation of new (advanced) call graph algorithms or as the foundation for analyses that require a call graph as an input. The docker container enables other researchers to replicate the evaluation from the paper. Furthermore, it makes it possible to compute call graphs for new libraries which is of particular interest for analyses related to Java libraries which are build upon call graphs.

Due to the incremental design of docker it is possible that other researchers build on our container. This enables custom evaluations using a well-defined reference environment.

4. USABLE

The list of dead-methods is a simple HTML file which lists the respective methods. The source code is provided along with a build script to compile the code and to execute the test suite. The build script also automatically downloads all necessary libraries. This makes it possible to directly build OPAL after checkout. Additionally, the code is generally well documented (https://www.openhub.net/p/OPAL-Project). This provides an excellent foundation for the implementation of new analyses or enhanced call graph algorithms.

[1] https://opensource.org/licenses

[2] http://opal-project.de/artifacts/index.php.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Call Graph Construction for Java Libraries using OPAL

Michael Reif Michael Eichberg Ben Hermann Johannes Lerch Mira Mezini
Technische Universität Darmstadt
Darmstadt, Germany
{lastname}@cs.tu-darmstadt.de

1. ARTIFACT EVALUATION

This manual shall guide you through the necessary steps to reproduce the results in the paper's evaluation. Though it is possible to build OPAL on one's own, it is recommended to use our docker container. It contains a pre-build version of OPAL that was used as the foundation for the paper. The proposed artifact (≈ 2GB) is publicly available on DockerHub[1].

2. SETUP THE ARTIFACT

Our artifact is shipped using docker, thus, it will run on every machine where docker[2] is installed.The container was tested using Windows 10 and MacOS X 10.11. Docker was configured with 4GB of RAM and 8 cores. Given an installation of docker, it is then possible to use the command line tools to download the container:

```
$docker pull mreif/fse2016:evaluation
```

3. HOW TO USE IT?

After pulling the container it can be run using:

```
$docker run -ti mreif/fse2016:evaluation
```

From now on you have multiple options. The first option is to reproduce the results of our evaluation. The second option is to generate statistics related to arbitrary call graphs, e.g., the number of call edges or entry points of a call graph.

Once the container is started you will find yourself in the directory of the evaluation project.

3.1 Reproducing the evaluation

In case you want to reproduce the paper's evaluation, you have to use the following command:

```
sbt run
Output:
[1] CallBySignatureCountEvalationAnalysis
[2] EvaluationStarter
```

This will run the current project using `sbt` and brings up a menu with two different options (classes with main methods that are in the current project). To reproduce the evaluation you have to run the second program (enter number `2`). This script may run – depending on your available resources – up to 1 hour. The generated data derived from all constructed call graphs will be written to the file `/home/libcg/output/results.txt`. Do the following to copy the file to your host system for more comfortable analysis:

```
$docker ps -alq
OUTPUT: <container-id>
$docker cp
| <container-id>:/home/libcg/output/results.txt
| <path-on-your-system>
```

3.2 Generate custom call graph numbers

If a custom JAR file shall be analyzed you have to download the JAR file of the target library as well as all necessary dependencies. To retrieve them, use the `wget` tool as follows:

```
wget -P /home/libcg/customLibraries/ <url to jar>
```

Before running the analysis, make sure that you are in the directory of the evaluation project (/home/libcg/evaluation). You can then type `sbt` to start the `sbt` console. If you need help in specifying the parameters you can enter the `run help` command. A custom analysis would look like:

```
run -cp="/home/libcg/customLibraries/junit-4.12.jar"
| -libcp="/usr/lib/jvm/java-8-openjdk-amd64/jre/lib"
| -analysisMode=library_with_closed_packages_assumption
```

The custom JAR will be analyzed now. The output will be printed on the console. Note that the entry points of the call graph will be calculated from the projects on the class path (`-cp` parameter) and that the libraries specified using `-libcp` are only used to build the overall class hierarchy. The different assumptions under which the call graphs can be build can be passed via the `analysisMode` parameter. All available modes are shown in the help command of the analysis.

[1] https://hub.docker.com/r/mreif/fse2016/
[2] https://www.docker.com/

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235