Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs (Artifact)

Madhurima Chakraborty
University of California, Riverside, CA, USA

Renzo Olivares
University of California, Riverside, CA, USA

Manu Sridharan
University of California, Riverside, CA, USA

Behnaz Hassanshahi
Oracle Labs, Brisbane, Australia

Abstract

Building sound and precise static call graphs for real-world JavaScript applications poses an enormous challenge, due to many hard-to-analyze language features. Further, the relative importance of these features may vary depending on the call graph algorithm being used and the class of applications being analyzed. In this paper, we present a technique to *automatically* quantify the relative importance of different root causes of call graph unsoundness for a set of target applications. The technique works by identifying the dynamic function data flows relevant to each call edge missed by the static analysis, correctly handling cases with multiple root causes and inter-dependent calls. We apply our approach to perform a detailed study of the recall of a state-of-the-art call graph construction technique on a set of framework-based web applications. The study yielded a number of useful insights. We found that while dynamic property accesses were the most common root cause of missed edges across the benchmarks, other root causes varied in importance depending on the benchmark, potentially useful information for an analysis designer. Further, with our approach, we could quickly identify and fix a recall issue in the call graph builder we studied, and also quickly assess whether a recent analysis technique for Node.js-based applications would be helpful for browser-based code. All of our code and data is publicly available, and many components of our technique can be re-used to facilitate future studies.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases JavaScript, call graph construction, static program analysis

Digital Object Identifier 10.4230/DARTS.8.2.7

Funding This research was supported in part by a gift from Oracle Labs and by the National Science Foundation under grant CCF-2007024. This research was partially sponsored by the OUSD(R&E)/RT&L and was accomplished under Cooperative Agreement Number W911NF-20-2-0267. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL and OUSD(R&E)/RT&L or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

Related Article Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi, "Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs", in 36th European Conference on Object-Oriented Programming (ECOOP 2022), LIPIcs, Vol. 222, pp. 3:1–3:28, 2022. https://doi.org/10.4230/LIPIcs.ECOOP.2022.3

Related Conference 36th European Conference on Object-Oriented Programming (ECOOP 2022), June 6–10, 2022, Berlin, Germany

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2022 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

The artifact supports the following contributions from the original article [1]:

- We present a novel approach to quantifying the importance of language features causing low recall in JavaScript call graphs. The approach properly handles missing call graph edges with multiple root causes, and also inter-dependent calls, where an edge is missing due to the absence of another edge.
- We describe implementations of a dynamic call graph and dynamic flow trace analysis of function values for JavaScript, both of which handle several hard-to-analyze JavaScript features.
- We present results and key observations from applying our techniques for the ACG algorithm [2] and a suite of framework-based web applications.

2 Content

The artifact package includes a VirtualBox VM image with all the dependencies already installed.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at: `https://doi.org/10.5281/zenodo.6541325`.

4 Tested platforms

Running the virtual machine requires an x86-64 system with at least 16GiB of RAM and the virtualization software VirtualBox.[1] The VirtualBox VM image contains all software requirements (code, data, and benchmarks) and has all dependencies resolved. For hardware support, please plan to allocate at least 12GiB RAM and 2 processors to the VM.

5 License

The artifact is available under license The Universal Permissive License (UPL), Version 1.0.

6 MD5 sum of the artifact

1eaa4ca32db3fae30f88df0963f950d8

7 Size of the artifact

7.89 GiB

[1] `https://www.virtualbox.org/`

8 Getting Started

To run the artifact virtual machine, import `JS-CG-RootCause-Quatification.ova` in VirtualBox and start the machine. The username and password for the VM are anon and 123456 respectively. Once the virtual machine has started, you can run all our experiments on the TodoMVC benchmark using our automation or manually.

8.1 Using our automation to run all the experiments and generate the data for TodoMVC

1. To generate the metrics and results before improvements to WALA (Figures 4-6 and 11-13, and Section 7):
   a. Please navigate to `/home/anon/WALA/com.ibm.wala.cast.js/src/main/` and replace the resources folder with the resources folder in `/home/anon/BeforeImprovementResources`.
   b. Next, create an output folder, say `/home/anon/test_output`, and copy-paste the file `/home/anon/analysis_results.json` (containing the application and framework files information about the todomvc frameworks) to the output folder.
   c. Once the output folder is ready open Terminal, publish the resource changes :

```
cd WALA
./gradlew publishToMavenLocal
```

   d. Now we are ready to navigate to jalangi2 directory where all of our scripts are located:

```
cd jalangi2
```

   e. Then, run our `todomvc_auto.py` script as instructed below to generate the optimistic and pessimistic metrics and results:

```
python3 experiments/todomvc_auto.py --todomvc-root=/home/anon/todomvc-
    master/ --path-to-experiments=/home/anon/jalangi2/ --path-to-wala-acg
    =/home/anon/WALA-ACG/ --path-to-wala-prop=/home/anon/WALA-prop-source/
     --outpath=/home/anon/test_output/ --chosen-framework=all --typ-scg=
    OPT
python3 experiments/todomvc_auto.py --todomvc-root=/home/anon/todomvc-
    master/ --path-to-experiments=/home/anon/jalangi2/ --path-to-wala-acg
    =/home/anon/WALA-ACG/ --path-to-wala-prop=/home/anon/WALA-prop-source/
     --outpath=/home/anon/test_output/ --chosen-framework=all --typ-scg=
    PES
```

   Note: Change the outpath to the output folder where you plan to store your outputs in case you are using a different folder from test_output. All other inputs should remain same.
   Note: In case the script fails for one or more benchmarks due to bandwidth issues, it may advise you to run the script on the those benchmarks (say vanillajs) separately. In such a case, change the `-chosen-framework` from `all` to the said benchmark name ex: `-chosen-framework=vanillajs`
   f. Once all results have been generated, execute the following scripts to consolidate the data in a more user friendly format:

```
python3 experiments/auto_csv_metrics.py /home/anon/test_output/
python3 experiments/auto_csv_results_recursive.py /home/anon/test_output/
    PessimisticResults/ /home/anon/test_output/ PES
python3 experiments/auto_csv_results_recursive.py /home/anon/test_output/
    OptimisticResults/ /home/anon/test_output/ OPT
```

Note: Change the outpath to the output folder where you have your outputs stored incase you are using a different folder from test_output.

g. The consolidated metrics and results data are in `/absolute/path/to/output/folder/` as `metrics_output.xlsx` and `<TypeofSCG>_root_causes_output.xlsx` respectively.
The raw data corresponding these experiments can be found under : `/home/anon/raw_output`
All graphs and charts (Figures 4,5,6,11,12,13) presented in the paper have been derived from this data.

2. To generate the metrics and results after improvements to WALA (corresponding to Section 7 and Figures 7, 8 and 10):

a. Please navigate to the `/home/anon/WALA/com.ibm.wala.cast.js/src/main/` and replace the resources folder with the resources folder in `/home/anon/AfterImprovementResources`.
b. Next, create another output folder say `/home/anon/test_output_improved` and copy+paste the following file `/home/anon/analysis_results.json` (containing the application and framework files information about the todomvc frameworks) to the output folder.
c. And, follow steps 1.3-1.6.
d. The raw data of the above experiments are present in `/home/anon/raw_output_improved`.
All graphs and charts (7,8,10) presented in the paper have been derived from this data.

Estimated runtime for the Automation with Optimistic data is 60 minutes and with Pessimistic data is 35 minutes with **32GB RAM**.

8.2 Step by Step Manual process to run all the experiments and generate the data for TodoMVC/any other benchmark

1. Generate the SCG for the benchmark:

```
cd WALA-ACG
./gradlew run --args="/absolute/path/to/todomvc/benchmark/ /absolute/path/to
    /output/folder/ $type-of-scg-OPT-for-Optimistic-or-PES-for-Pessimistic$"
```

2. Generate the properties flows for the benchmark:

```
cd WALA-prop-source
./gradlew run --args="/absolute/path/to/todomvc/benchmark/ /absolute/path/to
    /output/folder/"
```

3. Generate the DCG for the benchmark:

```
cd jalangi2
node --max-old-space-size=8192 experiments/pupServ.js /absolute/path/to/
    todomvc/benchmark/ /absolute/path/to/output/folder/DCG.json
```

4. Calculate the metrics:

```
node experiments/metrics/metric1.js /absolute/path/to/DCG/ /absolute/path/to
    /SCG /absolute/path/to/todomvc/benchmark/ /absolute/path/to/
    analysis_results.json $framework-name$
node experiments/metrics/metric2.js /absolute/path/to/DCG/ /absolute/path/to
    /SCG /absolute/path/to/todomvc/benchmark/ /absolute/path/to/
    analysis_results.json $framework-name$
```

```
node experiments/metrics/metric3.js /absolute/path/to/DCG/ /absolute/path/to
    /SCG /absolute/path/to/todomvc/benchmark/ /absolute/path/to/
    analysis_results.json $framework-name$
```

5. Compute the differences between the SCG and the DCG for the benchmark:

```
node experiments/metrics/StatVSDynDiffCallee.js /absolute/path/to/DCG /
    absolute/path/to/SCG/ /absolute/path/to/todomvc/benchmark/ /absolute/
    path/to/output/folder/
```

6. Generate the dynamic trace for the missing edges for the benchmark:

```
node --max-old-space-size=8192 experiments/pupServ2.js /absolute/path/to/
    todomvc/benchmark/ /absolute/path/to/output/folder/trace.json /absolute/
    path/to/static/calleeMap/
```

7. Compute the differences between the SCG and the DCG for the benchmark:

```
node --max-old-space-size=4096 experiments/parse_call.js /absolute/path/to/
    diff/file/ /absolute/path/to/dynamic/trace/ /absolute/path/to/todomvc/
    benchmark/ /absolute/path/to/static/flow/graph/ /absolute/path/to/static
    /call/graph/ /absolute/path/to/formatted/dynamic/call/graph $type-of-scg
    -OPT-for-Optimistic-or-PES-for-Pessimistic$"
```

8. Find the root causes for the benchmark:

```
node experiments/count.js /absolute/path/to/output/folder/$benchmark$_causes
    .json /absolute/path/to/output/folder/
```

9. Find the dynamic property reasons for the benchmark:

```
node experiments/findPropReasons.js /absolute/path/to/output/folder/
    StaticProps.json /absolute/path/to/output/folder/$benchmark$.json /
    absolute/path/to/output/folder/
```

References

1 Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic root cause quantification for missing edges in JavaScript call graphs (extended version). *CoRR*, 2022. URL: https://arxiv.org/abs/2205.06780.

2 Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *International Conference on Software Engineering*, ICSE, pages 752–761, 2013.