Who Are You not gonna Call?
A Definitive Comparison of Java Static Call Graph Creator Tools

Edit Pengő, Zoltán Ságodi, Ervin Kóbor

Abstract: Call graphs are fundamental for advanced, interprocedural control flow and data flow analysis tasks. In dynamic languages like Java, which allows polymorphism and reflection constructing a static call graph can be complicated. A missing or imprecisely connected edge might misguide the following algorithms causing errors in the overall analysis.

In this paper, we have collected six static analyzer tools for Java and performed a qualitative comparison on the call graph they generate. As part of the comparison, we introduced a method for pairing different notations of the same functions. We evaluated the collected tools on three open-source Java projects and on a small example containing most of the relevant Java language features. The results revealed several language structures that were handled differently by the static analyzers, which led to a difference in the created call graphs as well.

Keywords: Java, Call Graph, Static Code Analysis

Introduction

Developing quality software is a complex task and the unclear requirements, the underestimated efforts, and the strict deadlines make it even more difficult. Therefore, every software contains bugs even after its release. Static code analyzer tools help programmers produce more maintainable code and eliminate flaws early on by automatically analyzing the source code and identifying the potentially problematic places. However, the capabilities of these tools can differ considerably depending on the complexity of the representations and algorithms they use. Call graphs are building blocks for, for example, static control flow and data flow analysis. Therefore, it is crucial for the subsequent analyses to have a carefully constructed call graph especially if we consider dynamic language constructions as function pointers and polymorphism. In this work, we compared six open source static Java analyzer tools based on the call graphs we generated with them. We used three real life Java projects as test inputs along with a sample program containing the very essence of Java language. We introduced a comparing mechanism for the various graph representations provided by the tools and studied the results to identify Java language constructions that usually cause differences among the call graphs.

Murphy et al. [6] carried out a similar study of five static call graph creator for C in 1996. They identified significant differences in how the tools handled typical C constructs like macros. Rountev [9] built a framework to analyze differences in static and dynamic call chains in Java. They constructed static call chains from the edges of static call graphs. Reif et al. [8] studied the usability of call graph creation algorithms for Java libraries. They showed that there can be significant differences in the graphs depending on which algorithm was used. Lhoták proposed the importance of comparability among static analyzer tools. In his 2007 study [4] he built a general framework to compare static and dynamic call graphs, discussed the challenges of comparison and presented an algorithm to find the causes of differences in call graphs. There are several studies about dynamic call graph based fault detection like the work of Eichinger et al. [3] who created and mined weighted call graphs to achieve more precise bug localization. Liu et al. [5] constructed behavior graphs from dynamic call graphs to find noncrashing bugs and suspicious code parts with a classification technique.

The rest of the paper is organized as follows. Section gives a general description about call graphs and the tools we compared. The results of the comparison is presented in Section . Finally, we conclude the study in Section .

Call graph

Call graphs represent control flow relationships among the functions of a program. The nodes of the graph are the functions that are connected with directed edges. An edge from node *a* to node *b* indicates that function *a* invokes function *b*. Two types of call graphs can be distinguised: *static* and *dynamic* call graphs. Static call graphs are composed during the static analysis of the code. Ideally, they are conservative in a way that they contain every possible function call that can be realized during the execution of the program. Considering dynamic language structures like function pointers and polymorphism, it is clear that constructing static call graphs is not a trivial task. Code analyzers can implement several algorithms [2, 10, 1] that address the difficulties of dynamic linking and make the static call graph more precise. Dynamic call graphs are constructed from call traces recorded during a concrete execution of the program, therefore they are the subsets of the ideal static call graphs. Although they contain proper dynamic binding information, they show only those functions and calls that were used during execution.

Tools

First, we examined many Java static analyzer tools that could generate or could be modified to generate call graphs. Although many seemed promising, a lot of them had to be eliminated for various reasons. We searched for free, widely available, open-source programs that were robust enough to analyze complex, real-life Java systems. Countless plug-in based call graph visualizers are available in IDEs like Eclipse, however as their output is mostly visual they were not usable for our purposes. Finally, we have selected six tools for our comparison.

- *OpenStaticAnalyser*[1] (OSA) is an open-source multi-language deep static analyzer framework developed by Department of Software Engineering, University of Szeged. It calculates source code metrics, detects code clones and finds coding rule violations in the source code. We extracted the static call graph by extending OSA with a visitor.
- *Soot*[2] is a language manipulation and optimization framework developed by the Sable Research Group at the McGill University. Although its latest official release was in 2012, the project is active with regular commits and nightly builds.
- *Spoon*[3] is an open-source library for the analysis and transformation of Java source code [7]. The project is well documented and actively developed with Java 9 support as well. We expanded it with a visitor for the Java metamodel it provides to generate call graphs.
- *WALA* [4] is a static analyzer for Java bytecode and JavaScript. It was originally developed at the IBM T.J. Watson Research Center and it is still maintained as an open source project. Wala has a built-in call graph generator functionality that we feed with all the methods of the program as entry points.
- *Java Call Hierarchy Printer*[5] (CHP) is a spoon based method hierarchy printer. It is a small GitHub project with only one contributor which has seemingly been inactive since 2015. To gain call graphs, we have created a wrapper tool that feeds CHP with the method names of the analyzed code and processes the printed call hierarchies.
- *Gousiosg call graph builder* [6] is an Apache BCEL based Java Call Graph Utility for generating static and dynamic call graphs. The project has two contributors and the last commit was made in 2017. It works on jar files and produces the output in a simple text format that we processed with a wrapper program.

Results

The examined six tools gave different outputs where not only the file format differed but also the names of the Java methods did as well. In order to be able to compare the results, the different notations of the methods names had to be unified. It was an easy task for "basic" methods, but it was very difficult for coding-features like *inner*, *anonymous*, *generic classes* and *lambdas*. For example, an anonymous class inside an anonymous class had different names (`Example$1$1` and `Example$2`). Therefore we developed a common method representation and transformed every method name to it. To validate the transformation we created a sample Java program that contained most of the various class and method constructs Java 8 allows and manually verified the results. We executed the six tools on the sample project and first compared how similar the found methods are. Table 1 shows the results where the diagonal cells present the number of methods found by the given tool and every other cell in a row shows how many percent of its methods was found by the other tool. The tools found almost the same number of methods and only the constructors or initializations differed except CHP who could not properly

[1]OpenStaticAnalyser GitHub Page: https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer
[2]Sable/Soot GitHub Page: https://github.com/Sable/soot
[3]Spoon HomePage: http://spoon.gforge.inria.fr
[4]Wala HomePage: http://wala.sourceforge.net/wiki/index.php/Main_Page
[5]Call Hierarchy Printer GitHub Page: https://github.com/pbadenski/call-hierarchy-printer
[6]gousiosg/java-callgraph GitHub Page: https://github.com/gousiosg/java-callgraph

handle lambdas and generic classes either. Our method unification gave good results because, apart from CHP, at least 75% of the methods can be paired.

	Soot	OSA	Spoon	CHP	Gous.	WALA
Soot	**64**	85.9%	85.9%	70.3%	89.0%	90.6%
OSA	77.5%	**71**	95.8%	71.8%	85.9%	87.3%
Spoon	77.5%	95.8%	**71**	71.8%	85.9%	87.3%
CHP	75.0%	85.0%	85.0%	**60**	83.3%	80.0%
gous.	83.8%	89.7%	89.7%	73.5%	**68**	91.2%
WALA	79.5%	84.9%	84.9%	65.8%	84.9%	**73**

Table 1: Common methods in the sample proj.

	Soot	OSA	Spoon	CHP	Gous.	WALA
Soot	**267**	44.2%	44.2%	28.5%	47.6%	55.8%
OSA	80.3%	**147**	98.6%	57.1%	87.1%	87.1%
Spoon	79.2%	97.3%	**149**	56.4%	85.9%	85.9%
CHP	56.7%	60.4%	60.4%	**139**	59.7%	57.6%
gous.	79.9%	80.5%	80.5%	52.2%	**159**	84.3%
WALA	86.6%	74.4%	74.4%	46.5%	77.9%	**172**

Table 2: Common calls in the sample proj.

Comparison of Call Edges

After unifying method names, we were able to compare calls. Table 2 presents the comparison results of the calls on the sample project where a diagonal contains the number of calls the given tool found and the other cells in its row show how many percent of its calls were found by the other tools. As we can see, all tools but Soot identified more or less the same number of calls because Soot connects every instantiation to `Object` initialization, therefore every `new` expression yields a calls. Therefore, its low comparison values are expected as well because their maximum values (when all methods found by the other tools could have been paired) are around 60%. CHP also had worse results because it could not connect methods of anonymous classes to the right caller. On the other hand, the other OSA, Spoon and Gousiosg found very similar calls and the different handling of static and dynamic initializations and constructor calls caused the most difference.

Table 3 presents the results of the three examined projects: Joda 2.9.9 (85,911 KLOC), Apache Commons Math 3.6.1 (208,876 KLOC) and the Java ASG module of OpenStaticAnalyzer (44,453 KLOC). The results of CHP is missing because it did not work on larger systems. OSA, Spoon and Gousiosg found similar number of calls while Soot and WALA idetified much more calls. Besides, the three tools handle the "special calls" (e.g. the initializations and system classes) quite a similar way, therefore their calls also coincide. Since Gousiosg identifies more calls its results seem worse but among its calls can be found most of the calls OSA and Spoon found (see gous. column). The calls of Soot and WALA not only differ from the previous ones, but their calls do not correspond with each other. Its main reasons are that Soot includes every `Object` initialization and possible inherited classes while WALA connects almost every method from standard packages which are filtered out by the other tools. On the other hand, in case of OpenStaticAnalyzer the result of Soot is much closer to the other tools because in this system there were only several "special" (e.g. lambda) classes.

Conclusion

We collected six open-source Java analyzers that were able to or could be extended to generate call graph. To be able to compare the tools, we developed a program that unified the different method names the tools used and compared the call graph as well. We evaluated the six tools on a sample project and on three open-source systems and found that CHP did not work on large systems while only three out of the five tools gave similar results.

In the future, we plan to perform a more detailed analysis to better understand the differences of the call graphs. For this, we will utilize a graph database to efficiently use general graph similarity algorithms for a deeper comparison.

Acknowledgements

This research was supported by the project "Integrated program for training new generation of scientists in the fields of computer science", no EFOP-3.6.3-VEKOP-16-2017-0002. The project has been supported by the European Union and co-funded by the European Social Fund.

	Soot	OSA	Spoon	Gous.	WALA	Soot	OSA	Spoon	Gous.	WALA	Soot	OSA	Spoon	Gous.	WALA
Soot	**27,086**	12.7%	13.2%	13.6%	53.0%	**42,214**	8.3%	8.5%	9.4%	33.2%	**69,481**	13.3%	13.4%	13.2%	49.7%
OSA	34.3%	**9,995**	99.9%	86.9%	75.0%	17.5%	**20,059**	94.8%	84.9%	78.0%	67.5%	**13,643**	100.0%	91.2%	85.9%
Spoon	35.1%	98.0%	**10,189**	87.0%	75.4%	17.5%	92.7%	**20,494**	86.9%	74.2%	67.7%	99.0%	**13,775**	91.2%	85.1%
gous.	27.4%	64.5%	65.7%	**13,482**	53.6%	11.2%	48.0%	50.2%	**35,476**	45.3%	35.6%	48.5%	48.9%	**25,666**	44.5%
WALA	36.2%	18.9%	19.4%	18.2%	**39,657**	18.3%	20.5%	19.9%	21.0%	**76,496**	67.7%	23.0%	23.0%	22.4%	**50,990**

Table 3: Calls and common calls of Joda, Apache Commons Math and OpenStaticAnalyzer

References

[1] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. *SIGPLAN Not.*, 31(10):324–341, October 1996.

[2] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 – Object-Oriented Programming, 9th European Conference, Åarhus, Denmark*, pages 77–101. Springer Berlin Heidelberg, 1995.

[3] Frank Eichinger, Klemens Böhm, and Matthias Huber. Mining Edge-Weighted Call Graphs to Localise Software Bugs. In *Machine Learning and Knowledge Discovery in Databases*, pages 333–348, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] Ondřej Lhoták. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42, 2007.

[5] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S. Yu. Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. In *SDM*, 2005.

[6] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191, April 1998.

[7] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[8] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call Graph Construction for Java Libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 474–486, New York, NY, USA, 2016. ACM.

[9] Atanas Rountev, Scott Kagan, and Michael Gibas. Static and Dynamic Analysis of Call Chains in Java. *SIGSOFT Softw. Eng. Notes*, 29(4):1–11, July 2004.

[10] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical Virtual Method Call Resolution for Java. *SIGPLAN Not.*, 35(10):264–280, October 2000.