Systematic Comparison of Six Open-source Java Call Graph Construction Tools

Judit Jász[a], István Siket[b], Edit Pengő[c], Zoltán Ságodi[d] and Rudolf Ferenc[e]
University of Szeged, Department of Software Engineering, Árpád tér 2., H-6720 Szeged, Hungary

Keywords: Java, Call Graph, Static Analysis, Tool Comparison.

Abstract: Call graphs provide the groundwork for numerous analysis algorithms and tools. However, in practice, their construction may have several ambiguities, especially for object-oriented programming languages like Java. The characteristics of the call graphs – which are influenced by building requirements such as scalability, efficiency, completeness, and precision – can greatly affect the output of the algorithms utilizing them. Therefore, it is important for developers to know a well-defined set of criteria based on which they can choose the most appropriate call graph builder tool for their static analysis applications. In this paper, we studied and compared six static call graph creator tools for Java. Our aim was to identify linguistic and technical properties that might induce differences in the generated call graphs besides the obvious differences caused by the various call graph construction algorithms. We evaluated the tools on multiple real-life open-source Java systems and performed a quantitative and qualitative assessment of the resulting graphs. We have shown how different outputs could be generated by the different tools. By manually analyzing the differences found on larger programs, we also found differences that we did not expect based on our preliminary assumptions.

1 INTRODUCTION

Producing high-quality software is an important requirement of today's industrial development, so, naturally, there are many tools and methodologies available to aid quality management. A subset of these tools are static source code analyzers that help programmers eliminate flaws and rule violations early on by automatically analyzing the subject system and highlighting its potentially erroneous parts. However, their capabilities can differ significantly depending on the complexity of the internal representations and algorithms they use.

Call graphs are directed graphs representing control flow relationships among the methods of a program. The nodes of the graph denote the methods, while an edge from node *a* to node *b* indicates that method *a* invokes method *b*. Call graphs can either be considered static or dynamic depending on whether they were constructed during static or dynamic analysis. Static graphs tend to overestimate the accurate call graphs, while the quality and precision of the dynamic graphs are heavily influenced by the size and quality of the corresponding test-suite. In this work we are considering static call graphs only.

As call graphs are the main building blocks for modeling interprocedural control and data flow, their soundness can greatly affect the results of subsequent analyses. Developers need to carefully consider how their call graphs are constructed before they incorporate them into a novel algorithm. In the case of object-oriented languages, the target of a call often depends on the runtime behavior of the program, therefore, a static call graph builder has to make assumptions about what methods could be called, resulting in possible imprecisions. Call graph builder algorithms addressing this challenge have an extensive literature, including detailed comparisons (Lhoták, 2007), (Tip and Palsberg, 2000), (Grove et al., 1997), (Murphy et al., 1998), (Grove and Chambers, 2001), (Lhoták and Hendren, 2006). However, there are other factors that influence the structure of a call graph as well, for example, the handling of different kinds of initializations or anonymous classes. In this paper, we aimed to study these factors through the evaluation of six static call graph builder tools for Java. The following

[a] https://orcid.org/0000-0001-6176-9401
[b] https://orcid.org/0000-0003-4064-1489
[c] https://orcid.org/0000-0002-4500-8693
[d] https://orcid.org/0000-0001-5828-6265
[e] https://orcid.org/0000-0001-8897-7403

Jász, J., Siket, I., Pengő, E., Ságodi, Z. and Ferenc, R.
Systematic Comparison of Six Open-source Java Call Graph Construction Tools.
DOI: 10.5220/0007929201170128
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 117-128
ISBN: 978-989-758-379-7

Research Questions (RQs) guided the direction of our research:

- **RQ1:** How does the different handling of Java's language features affect the resulted call graphs?
- **RQ2:** How different could the call graphs be in practice?
- **RQ3:** Do we get the same graphs if we ignore the known differences?

We constructed an example code – full of language features that we expected to challenge the tools (available as an online appendix, see Section 4) – and compared the resulting graphs. We also performed an evaluation on four real-life Java systems in order to study the differences on a bigger scale. The results of the Maven[1] and ArgoUML[2] projects are presented in this paper, while the results of the other projects are available as part of the online appendix.

The rest of the paper is organized as follows. Section 2 provides a brief background on call graphs and also discusses the related literature. The six tools we compare are described in Section 3. We define the steps of our evaluation process in Section 4 and analyze the results quantitatively and qualitatively in Section 5. Finally, threats to the validity of our results are examined in Section 6 before we draw our conclusions in Section 7.

2 RELATED WORK

Call graphs are the basis of many software analysis algorithms, such as control flow analysis, program slicing, program comprehension, bug prediction, refactoring, bug-finding, verification, security analysis, and whole-program optimization (Weiser, 1981), (Feng et al., 2014), (Christodorescu and Jha, 2003), (Wagner et al., 1994). The precision and recall of these applications depends largely on the soundness and completeness of the call graphs they use. Moreover, call graphs can be employed to visualize the high level control flow of the program, thus helping developers understand how the code works. There are several studies about dynamic call graph-based fault detection, like the work of Eichinger et al. (Eichinger et al., 2008), who created and mined weighted call graphs to achieve more precise bug localization. Liu et al. (Liu et al., 2005) constructed behavior graphs from dynamic call graphs to find non-crashing bugs and suspicious code parts with a classification technique.

[1]https://github.com/apache/maven

[2]http://argouml.tigris.org/

Regardless of whether the examined language is low-level and binary or high-level and object-oriented, call graph construction can always lead to some difficulties (Bacon and Sweeney, 1996), (Reif et al., 2016). A call graph is accurate if it contains exactly those methods and call edges that might get utilized during an actual execution of the program. However, in some cases, these can be hard to calculate. For example, if several call targets are possible for a given call site, then deeper examination is needed to determine which ones to connect as precisely as possible. This examination can be done in a context-dependent or context-independent manner; naturally, the choice influences the generated call graph. Context-dependent methods are more accurate in return for greater resource usage. To mitigate the resource demands of such methods, the analysis of the programs often only starts from the `main` method or a few entry points instead of starting from every method. This might result in a less accurate call graph. To improve the accuracy of context-independent methods, the following algorithms can be used for object-oriented languages: *Class Hierarchy Analysis (CHA)* (Dean et al., 1995), *Rapid Type Analysis (RTA)* (Bacon and Sweeney, 1996), *Hybrid Type Analysis (XTA)*(Tip and Palsberg, 2000), *Variable Type Analysis (VTA)* (Sundaresan et al., 2000).

Another important question during call graph creation is the handling of library calls (Ali and Lhoták, 2012). Including library calls not only makes the call graph bigger, it also requires the analysis of the libraries which can be quite resource consuming. However, the exclusion of library elements may cause inaccuracies when developers implement library interfaces or inherit from library classes. The analysis of library classes might involve private, inaccessible methods as well. Michael Reif et al. (Reif et al., 2016) discussed the problem that the often used call graph builder algorithms, such as *CHA* and *RTA*, do not handle libraries separately according to their availability. The recommended algorithm in this work reduces the number of call edges by 30%, in contrast to other existing implementations. The tools we selected for our comparison represent library calls and library methods at various levels of detail.

As mentioned in Section 1, many comparative studies are available about call graph creation. Grove et al. (Grove et al., 1997) implemented a framework for comparing call graph creation algorithms and assessed the results with regard to precision and performance. Murphy et al. (Murphy et al., 1998) carried out a study similar to ours about the comparison of five static call graph creators for C. They identified significant differences in how the tools handled

typical C constructs like macros. Hoogendorp gave an overview of call graph creation for C++ programs in his thesis (Hoogendorp, 2010). Antal et al. (Antal et al., 2018) conducted a comparison on JavaScript static call graph creator tools. Similarly to our work, they collected five call graph builders and analyzed the handling of JavaScript language elements and the performance as well. As a result, they provided the characterization of the tools that can help in selecting the one, which is most suitable for a given task. Tip et al. (Tip and Palsberg, 2000) tried to improve the precision of *RTA* by introducing a new algorithm. On average, they reduced the number of methods by 1.6% and the number of edges by 7.2%, which can be a considerable amount in the case of larger programs. Lhoták (Lhoták, 2007) compared static call graphs generated by Soot (Sable Research Group, 2019) and dynamic call graphs created with the help of the *J (Sable *J, 2019) dynamic analyzer. He built a framework to compare call graphs, discussed the challenges of the comparisons, and presented an algorithm to find the causes of the potential differences in call graphs.

Reif et al. (Reif et al., 2018) dealt with the unsoundness of Java call graphs. They compared the call graph creator capabilities of two analyzer tools, WALA and Soot. They evaluated different configurations of the tools on a small testbed. Their main goal was to decide whether a tool handles a specific language element or not, and - unlike our work - did not investigate the way it is handled. An assessment suite for the comparison of different call graph tools is proposed as well. Our work is similar to their study, however, we performed an in-depth examination to identify what differences can occur between call graph builder tools. The six tools we selected for our research have various properties and ways of analysis, for example, there are both source- and bytecode analyzers, while Reifs et al. only analyzed bytecode based tools. As a result, we provide a full scale of factors that can cause ambiguities in the call graph creation. There are obvious factors, like the handling of polymorphism and library calls that were analyzed before. However, we highlight other, less evident aspects as well that have to be considered before using or developing a call graph builder tool.

3 CALL GRAPH CONSTRUCTION TOOLS

We studied numerous static analyzer tools for Java to decide whether they could generate – or could be easily modified to generate – call graphs. We searched for widely available, open-source programs from recent years, which could analyze complex, real-life Java systems. We discarded many plug-in-based tools, as they produced only a visual output (e.g., CallGraph Viewer (CallGraphViewer, 2019)), while other promising candidates were not robust enough on larger systems (e.g., Java Call Hierarchy Printer (Badenski, 2019)). In some cases, the call graphs had to be extracted directly from the inner representation of the analyzer. However, we eliminated any tool that did not provide enough information to reconstruct the caller-callee relationships between compilation units without major development (e.g., JavaParser (Danny van Bruggen, Federico Tomassetti, Nicholas Smith, Cruz Maximilien, 2019)).

The description of the six tools that met our selection criteria is presented below.

Soot (Sable Research Group, 2019) is a widely used language manipulation and optimization framework developed by the Sable Research Group at the McGill University. It supports analysis up to Java 9 and works on the compiled binaries. Although its latest official release was in 2012, the project is still active on GitHub, from where we acquired the 3.2.0 release, which was the latest version at the time. Soot has a built-in call graph creator functionality that can be parameterized with multiple algorithms. We employed the *CHA* algorithm during construction.

OpenStaticAnalyzer (OSA) (DSE University of Szeged, 2019) is an open-source, multi-language static analyzer framework developed by the University of Szeged. It calculates source code metrics, detects code clones, performs reachability analysis, and finds coding rule violations in Java, JavaScript, Python, and C# projects. Besides the recursive directory-based analysis of the source code, OSA is also capable of wrapping the build system (*maven* or *ant*) of the project under examination. This can make the analysis more precise, as generated files will be handled as well. We extracted the call graph of our project by traversing its Abstract Syntax Tree [3] (AST) like internal representation and collecting every available invocation information.

SPOON (Pawlak et al., 2015) is an open-source, feature-rich Java analyzer and transformation tool for research and industrial purposes. It is actively maintained, supports Java up to version 9, and while several higher-level concepts (e.g., reachability) are not provided ”out of the box”, the necessary infrastructure is accessible for users to develop their own. SPOON performs a directory analysis of the source code and builds an AST-like metamodel, which is the

[3] Abstract Syntax Tree represents the syntactic structure of the source code in a hierarchical tree-like form.

basis for these further analyses and transformations. Similarly to the above mentioned OSA implementation, the call information can be obtained by processing the AST-like inner representation of SPOON. The library is well-documented and provides a visual representation of its metamodel, which helped us thoroughly study its structure. We used the 7.0.0 version for our research.

Java Call Graph (JCG) (Georgios Gousios, 2019) is an Apache BCEL (Apache Commons, 2019) based utility for constructing static and dynamic call graphs. It can be considered a small project, as it has only one major contributor, Georgios Gousios, whose last commit (at the time of writing) is from October, 2018. It supports the analysis of Java 8 features and requires a `jar` file as an input. A special feature of the analyzer is the detection of *unreachable* code. As a result, the call graph does not include calls from code segments that are never executed.

WALA (WALA, 2019) is a static and dynamic analyzer for Java bytecode (supporting syntactic elements up to Java 8) and JavaScript. Originally, it was developed by the IBM T.J. Watson's Research Center; now it is actively developed as an open-source project. Similarly to Soot, it also has a built-in call graph generation feature with a wide range of graph building algorithms. We used the *ZeroOneContainerCFA* graph builder for our research, as it performs the most complex analysis. It provides an approximation of the Andersen-style pointer analysis (Andersen, 1994) with unlimited object-sensitivity for collection objects. The generator had to be parametrized with the entry points, from which the call graphs would be built. To make the results similar to the results of the other tools, we treated all not-private, non-abstract methods as entry points (instead of just the `main` methods). For other configuration options, we used the default settings provided in the documentation and example source codes.

Eclipse JDT (Eclipse JDT, 2019) is one of the main components of the Eclipse SDK (Eclipse, 2019). It provides a built-in Java compiler and a full model for Java sources. We created a JDT based plugin for Eclipse Oxygen that supports even Java 10 code, to extract the call graph from the extensive, AST-like inner representation.

4 EVALUATION PROCESS

In Java, methods can be distinguished by fully qualified names, which include the package name, the class name, the name of the method, and the list of the parameter types. These methods can be referred by their name and by the appropriate parameter types. However, the nomenclature of some language elements is not standardized, for example, the naming of the anonymous classes and methods, or the notations of lambda expressions. Moreover, it is also possible that compiler-generated code parts are not present in some source-code based representations.

In applications, call graphs can only be used effectively if the call dependencies among the nodes solely cover real dependencies and also include all those that indicate any data or control dependencies. In order to compare the soundness and unsoundness of the call graphs generated by different tools, we need to identify the corresponding nodes – the targets of the potential invocations – in multiple graphs. Naturally, each tool produced a slightly different output. For example, OSA and WALA use the standardized naming convention[4], while others employ their own notation system. To illustrate, here are two different representations of the public void foo(String[] str) method:

- OSA: foo([Ljava/lang/String;)V
- Soot: void foo(java.lang.String[])

Despite the different textual forms, these two representations can be matched easily. We had to implement a specific graph loader for each tool to handle the aspects of its method naming convention. A method name unification algorithm was introduced to overcome all notational differences. However, two language features, the anonymous and generic code elements, needed extra consideration, therefore, the line information of the methods was also involved in the method pairing process. We have note that line information was not always available or reliable. The developmental and fine-tuning steps of the method name unification algorithm and validation of the pairing mechanism based on that were discussed in one of our previous articles (Pengő and Ságodi, 2019). We performed the graph comparisons on the unified graphs. We analyzed the kinds of nodes and edges that were found by each tool.

First, we performed a comparison on a small Java sample code (454 LOC) to identify how the tools handle different Java language elements. The code, through trivial test cases, helps highlight how the language features impact the generated call graphs. We tested the handling of polymorphism, reflection, lambda expressions, etc. with one simple example each, then manually studied the outputs. Native JNI calls and callback functionalities are not tested because their handling is far beyond the capabilities of

[4]Standard naming convention for Java methods: https://docs.oracle.com/en/java/javase/11/docs/specs/jni/intro.html

ordinary static analyzer tools. After the in-depth examination of the sample code we conducted an analysis of large, real-life Java projects in order to measure the impact of various handling procedures on a large scale.

The source of our tool, which compare the different outputs of the different call graphs, the used call graph tools, the example code and analyzed programs with the comparison results are available in our online appendix at http://www.inf.u-szeged.hu/~ferenc/papers/StaticJavaCallGraphs.

5 COMPARISON OF CALL GRAPHS

To answer our research questions, in this section we first characterize the language elements responsible for the variance of call graphs. We perform a qualitative and quantitative analysis, and finally we classify the differences of the resulted call graphs manually, so that we can learn more about the causes of the differences.

5.1 Handling of Language Features

In this subsection, we summarize the language elements that are handled differently, therefore, cause differences in the generated call graphs.

Initializer Methods. The handling of the different types of initializations is one of the main sources of differences. Naturally, all of the tools represent constructor calls. With the exception of JDT, all of them detect and connect generated default constructors even without the instantiation of an object, and derived classes' calls to super constructors are represented as well. In case of AST-based call graphs, initializer blocks and constructors have different nodes in the call graph. Bytecode based call graph builders represent such nodes as one. The initializer methods of nested classes also cause discrepancies in the graphs, because bytecode based tools (Soot, WALA and JCG) represent a reference to the outer class as an additional parameter in the parameter list. Obviously, source code based tools miss this parameter, since it is not present in the actual code. Both solutions are acceptable, and does not lessen the accuracy of calls, although it makes the node pairing more challenging.

Static initializer blocks are executed when a class is loaded by the class loader of the Java Virtual Machine. This is a dynamic process, triggered by different types of usage of a class, therefore, representing static initializer nodes in a static call graph can be cumbersome and incidental. All tools represent static initializer blocks, however, with different details and call edges. A large part of Soot's graphs are made up of these nodes. When a class is used and it has at least one static field declared, Soot inserts a corresponding static initializer node.

Polymorphism. Polymorphism is one of the most important traits of an object-oriented language. They occur most often when a parent class reference is used to refer to a child class object. However, polymorphism can cause inaccuracies in the call graphs, as static analyzers might be unable to decide whether an object reference is of its declared type or any subtype of its declared type. So, when a method is invoked, instead of linking the proper overridden method, most of the analyzers only link the parent method in the graph. This problem can be resolved by employing an algorithm that tries to approximate the call target.

Only WALA and Soot use an advanced algorithm, namely a type of points-to analysis, whilst the other tools rely on simple Name Based Resolution (NBR) (Tip and Palsberg, 2000). The NBR tools, such as OSA, SPOON, JCG, and JDT represent polymorphic methods with their static type. As there are many comparative studies about call graph builder algorithms (Lhoták, 2007), (Tip and Palsberg, 2000), (Grove et al., 1997), (Murphy et al., 1998), (Grove and Chambers, 2001), (Lhoták and Hendren, 2006), the thorough examination of the handling of polymorphism is not in the focus of our research. In our current evaluation, Soot uses the Class Hierarchy Analysis (CHA) to resolve the target of the polymorphic call. CHA makes the assumption that every overridden implementation of methods on a given inheritance hierarchy is callable at the call sites. In many cases, this will clearly result in false positive calling relationships, as we will see in the discussion of anonymous classes. The ZeroOneContainerCFA algorithm of WALA is more sophisticated, but the implementation is incorrect in the sense that neither WALA nor Soot realize method invocations of default methods of interfaces. If a method is not overridden in the derived class, JCG generates a copy of the base method in the derived class. This method is callable if the static type of the object at the calling site makes it possible, however, this method does not refer to the original method and the called methods of the original. So an application, which traverses the possible execution paths, will miss some potential paths.

Anonymous Source Code Elements. Anonymous methods and classes can cause difficulties in the node pairing process because there is no standard way of naming. If the tools do not provide valid line information, some anonymous methods will remain unmatched, which results in different call edges as well.

However, our examination revealed that this is not the most important reason for differences caused by the anonymous elements. The instantiations of inner classes and the calls of constructors work well in every tool. Although the naming of the inner classes can be different for each tool, the pairings of the corresponding nodes of the graphs are made feasible with the help of additional line information. According to the properties discussed during the polymorphism, it is not surprising, that OSA, SPOON, JCG are not able to invoke the methods overloaded by the anonymous classes. CHA algorithm of the Soot can be problematic. Since methods of the anonymous classes are not reachable in many contexts, it is not always correct to consider these to be the part of the hierarchy. In many cases, this can cause false positive call relations in the call graph.

Generic Elements. In order to implement generics, the Java compiler applies type erasure, where it replaces all type parameters in generic types with their bounds or with the `Object` if the type parameters are unbound. This type erasure is used by the call graph tools in most cases. Although WALA and Soot are also using type erasure to specify the target method, whose definition contains at least one generic parameter, these tools propagate the types of the actual parameters into the called method.

When using JDT, we collect information from the AST representation of the analyzed program, and we determine the target of a particular call with method binding information at the call site. If a generic method is instantiated with different types, we get more nodes in the call graph, which represent the same method. Of course this is our fault, since JDT provides all necessary information with which we could approach the accuracy of either WALA or Soot.

Java 8. Java 8 introduced the concept of functional interfaces. These are interface classes that contain exactly one abstract method. Lambda expressions and method references, which are also new features of Java 8, can be used to reference such functional interfaces. Since lambda expressions cannot, strictly speaking, be considered methods, their interpretation in a graph that represents methods as nodes is a bit cumbersome. Out of the six tools, only WALA creates dedicated nodes for lambda expressions, other tools represent them with the interface they implement. Similarly to lambda expressions, WALA also handles functional interfaces with specific nodes, to which the lambda and method reference nodes are connected. Although it would not be impossible (by tracing the inner calls of the Java libraries) we concluded from our research that call graph builder tools mostly fail to detect and, therefore, represent what actual methods are called through the mentioned calls.

Dynamic Method Calls. The reflection and the method handle mechanism of Java make it possible to determine the target of a method invocation dynamically in runtime. On our sample code we tested whether any of the six tools could determine the targets of a basic reflection and a method handle call. Since none of the tools provided a solution for the handling of these invocations, we are not dealing with them in the rest of the paper, similarly to native JNI calls and callback methods.

Our RQ1 was: "How does the different handling of Java's language features affects the resulted call graphs?" The handling of some language elements causes additional nodes to appear in the call graph. This is necessary in some cases (e.g. in the case of default constructors or generated methods), while in others it is only a technical help for call graph construction (e.g. linking inherited methods). In some cases there are many potential targets of a method call. In such situations, different call graph tools can have a different number of call edges from the given call site.

5.2 Quantitative Differences of Call Graphs

We summarized the results of our sample code in Tables 1 and 2. The numbers in the main diagonals of the tables are the number of methods (e.g. Soot found 114 methods, see Table 1) or calls (e.g. Soot identified 404 invocations, see Table 2) found by the tools, while the top left cell contains the number of distinct methods found by the different tools, i.e., it is the number of methods or calls in the union (the six tools together found 176 distinct methods and 472 different method invocations). The number of calls discovered by the individual tools ranges from 211 to 404. The percentages in the row of a tool show the ratio of its methods or calls that were found by the other tool presented in the given column. For example, WALA found 249 calls and 175 of them were found by JDT as well, which results in 70.28%. On the other hand, JDT detects 211 call edges from which WALA found 175 as well. However, as JDT has fewer calls than WALA, the ratio is higher, 82.94%. This means that the table is not symmetrical.

For an easier visual overview, the percentages above 80% are colored green, while the percentages below 60% are red. Tables 1 and 2 show that the results of OSA and SPOON are well aligned on the example code, OSA covers all the methods and edges of the SPOON's graph. SPOON connects three additional library methods into the graph, and this causes

Table 1: Common methods of the sample code.

176	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**114**	76.39%	78.47%	81.25%	82.64%	73.61%
OSA	92.44%	**119**	100.00%	95.80%	91.60%	94.12%
SPOON	92.62%	97.54%	**122**	95.90%	91.80%	94.26%
JCG	86.67%	84.44%	86.67%	**135**	85.19%	81.48%
WALA	93.70%	85.83%	88.19%	90.55%	**127**	82.68%
JDT	90.60%	95.73%	98.29%	94.02%	89.74%	**117**

Table 2: Calls of the sample code.

472	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**404**	51.73%	52.48%	53.96%	58.42%	43.56%
OSA	89.70%	**233**	100.00%	94.85%	89.27%	82.83%
SPOON	87.60%	96.28%	**242**	92.56%	87.19%	82.64%
JCG	87.20%	88.40%	89.60%	**250**	86.40%	74.80%
WALA	94.78%	83.63%	84.74%	86.75%	**249**	70.28%
JDT	83.41%	91.47%	94.79%	88.63%	82.94%	**211**

Table 3: Methods of the Maven project.

7,567	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**4,769**	33.84%	50.05%	55.19%	43.24%	53.66%
OSA	64.33%	**2,509**	77.40%	75.69%	57.83%	74.33%
SPOON	63.34%	51.76%	**3,748**	87.22%	44.18%	82.87%
JCG	68.36%	49.34%	84.91%	**3,849**	45.31%	87.63%
WALA	92.31%	64.94%	74.55%	78.09%	**2,236**	84.70%
JDT	61.38%	47.09%	75.58%	82.71%	45.60%	**4,239**

Table 4: Methods of the ArgoUML project.

28,987	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**14,905**	61.61%	66.51%	69.23%	31.02%	68.26%
OSA	50.61%	**18,148**	55.11%	56.11%	21.66%	57.95%
SPOON	86.19%	86.97%	**11,447**	92.93%	35.42%	91.05%
JCG	66.23%	65.28%	68.55%	**15,574**	26.97%	70.43%
WALA	96.74%	82.27%	85.36%	88.00%	**4,783**	87.10%
JDT	78.98%	82.35%	82.02%	85.97%	32.48%	**12,929**

the slight difference.

Having analyzed the sample code, we evaluated the tools on larger projects as well, namely the ArgoUML-0.35.1[5] and Maven-3.6.0[6]. ArgoUML is a UML modeling tool with 180 KLOC, while Maven is a library tool with 80 KLOC, and older versions of both are also presented in the Qualitas Corpus database (Tempero et al., 2010). Although this paper compares the call graphs of these two programs only, we repeated our evaluation on other tools too. The results of these measurements are also available in our online appendix at http://www.inf.u-szeged.hu/~ferenc/papers/StaticJavaCallGraphs.

Table 5: Calls of the Maven project.

70,192	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**63,839**	3.52%	6.29%	8.28%	6.00%	6.46%
OSA	47.95%	**4,684**	64.50%	63.86%	38.32%	61.38%
SPOON	56.24%	42.32%	**7,139**	85.28%	31.18%	80.25%
JCG	59.76%	33.83%	68.87%	**8,840**	32.34%	74.07%
WALA	99.92%	46.81%	58.04%	74.55%	**3,835**	53.17%
JDT	59.97%	41.81%	83.31%	95.22%	29.65%	**6,877**

Table 6: Calls of the ArgoUML project.

332,806	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**292,212**	8.10%	8.42%	8.77%	3.88%	8.58%
OSA	45.13%	**52,441**	49.96%	49.88%	13.25%	56.14%
SPOON	80.10%	85.26%	**30,730**	82.78%	24.03%	89.45%
JCG	63.80%	65.13%	63.33%	**40,163**	19.01%	71.11%
WALA	98.65%	60.48%	64.27%	66.45%	**11,491**	56.52%
JDT	71.89%	84.38%	78.79%	81.86%	18.62%	**34,888**

[5]http://argouml-downloads.tigris.org/source/browse/argouml-downloads/trunk/www/argouml-0.35.1/

[6]https://mvnrepository.com/artifact/org.apache.maven/maven-core/3.6.0

Table 3-Table 6 show the differences among the tools. As both examples show, Soot represents many more methods in both graphs. One likely reason for this is the detailed portrayal of static initializer nodes, and the representation of all overridden methods caused by CHA algorithm. On the opposite side, WALA contains fewer methods (and so edges) than all the others, thanks to its more precise pointer analysis, and to the fact that WALA only processes methods available from certain entries. We can also observe (which did not become apparent from our sample code) that OSA's methods often differ from those of the other tools'. The reason for this is that OSA's analysis is library based, meaning that it analyzes every library that is on the analysis's path, but can only resolve library references based on names or the actual type only, not through the declarations. In the case of polymorphic calls, this results in the creation of nodes that differ in name from the other corresponding nodes of other tools' graphs. Since, in most cases the called methods do not even have line information, pairing cannot be achieved. Moreover, OSA will not process library functions that other tools bring in through searching on the project class path.

In RQ2, we investigate the extent to which call graphs can differ in practice. As we can see based on the analyzed programs, we get considerably different graphs. Although our pairings are not correct in every case, as OSA's previous example showed, the number of edges and nodes clearly display the discrepancy. Even on the small example, one tool defines twice as many edges as the other (Table 2), but in case of the large projects, the number of edges differ even more considerably.

5.3 Examination of the Causes of Differences

The main question of this section is, whether we could get the same graphs if we eliminate all the known differences of the tools? A given application that is built on call graphs may demand different call graph properties. Is it possible to bring the tools' outputs closer together in order to really be able to compare the generated graphs, therefore, assist in finding a suitable tool or method for a particular call graph based application? If we ignore the known differences from the generated call graphs, do we get the same resulting

graph by each of the tools?

The structure of call graphs are influenced by the following three attributes of a call graph creator tool:

- How are the crucial language elements handled?
- What method is used in processing the data: does it use library analysis or some kind of pointer or reachability analysis?
- How does it deal with dynamic calls that cannot be resolved during static analysis. In other words, what kind of algorithm does it use to make the connected calls more accurate?

In this section, we are going to non-exhaustively demonstrate how much filtering out a small discrepancy can help in bringing graph results closer together. In each step, we create a subgraph from the original with the help of a given filtering mechanism and compare it to the call graph that was composed in the previous step (or with the original if it is the first step). The filtering is applied on the nodes. Naturally, the edges that are in connection with that node are eliminated as well. So, the results of the tools are getting closer together.

Table 7: Common calls of the Maven project after eliminating the clinit calls detected only by Soot (The number of investigated methods is 7,140.)

69,200	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**62,847**	3.57%	6.39%	8.41%	6.10%	6.56%
OSA	47.95%	**4,684**	64.50%	63.86%	38.32%	61.38%
SPOON	56.24%	42.32%	**7,139**	85.28%	31.18%	80.25%
JCG	59.76%	33.84%	68.87%	**8,840**	32.34%	74.07%
WALA	99.92%	46.81%	58.04%	74.55%	**3,836**	53.17%
JDT	59.97%	41.81%	83.31%	95.22%	29.65%	**6,877**

Table 8: Common calls of the ArgoUML project after eliminating the clinit calls detected only by Soot (The number of investigated methods is 28,489).

320,036	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**279,442**	8.47%	8.81%	9.17%	4.06%	8.98%
OSA	45.13%	**52,441**	49.96%	49.89%	13.25%	56.14%
SPOON	80.10%	85.26%	**30,730**	82.78%	24.03%	89.45%
JCG	63.80%	65.13%	63.33%	**40,163**	19.01%	71.11%
WALA	98.65%	60.48%	64.27%	66.45%	**11,491**	56.52%
JDT	71.89%	84.38%	78.79%	81.86%	18.62%	**34,888**

Eliminating Differences Caused by Language Elements. In Section 5.1 we saw that certain language elements can significantly increase the amount of nodes, and, therefore, edges a graph has, which can cause large differences. For example, every time a class with a static member was used, Soot attached a reference to that class's static initializer block. For this reason, many edges became part of the graph that in reality might not be executed. In the first step, we decided to filter out the static initializer nodes that appear in Soot's graphs in the following way: the static initializer nodes only have incoming edges in the graphs of Soot and do not appear in the other tools' graphs. Table 7 and 8 show how much closer the tools outputs get to each other after the elimination of the large number of static initializer nodes and their connections. Compared to Tables 5 and 6, the difference is observable only in the first row. It is obvious, as only Soot's connections were involved in the filtering process. In case of Maven, we detracted 992 edges from Soot's graph this way, 12,770 in the case of ArgoUML (the number of nodes for Maven is 62, for ArgoUML it is 498). We concluded that the first filtering step did not bring Soot that much closer to the other tools.

Table 9: Common calls of the Maven project after eliminating library calls (The number of investigated methods is 4,216).

7,530	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**4,139**	27.04%	53.52%	61.97%	33.97%	56.8%
OSA	45.23%	**2,474**	59.78%	58.57%	35.49%	59.58%
SPOON	54.34%	36.29%	**4,076**	85.35%	28.75%	83.02%
JCG	60.4%	34.12%	81.92%	**4,247**	31.29%	89.45%
WALA	99.93%	62.4%	83.3%	94.46%	**1,407**	83.58%
JDT	58.88%	36.91%	84.75%	95.14%	29.45%	**3,993**

Table 10: Common calls of the ArgoUML project after eliminating library calls (The number of investigated methods is 21,252).

59,215	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**34,574**	42.51%	41.15%	42.44%	16.39%	43.96%
OSA	43.63%	**33,685**	43.34%	48.78%	12.59%	53.82%
SPOON	87.48%	89.76%	**16,264**	82.56%	26.94%	91.98%
JCG	64.26%	71.96%	58.8%	**22,834**	17.76%	74.42%
WALA	100.00%	74.84%	77.29%	71.54%	**5,668**	71.33%
JDT	78.09%	93.14%	76.86%	87.3%	20.77%	**19,463**

Eliminating Algorithmic Differences. Often, it is not because of language elements that graphs become very different. Examining the nodes of the sample code, we noticed that the tools handle Java library calls in various ways. Some tools, like OSA, represent library calls with less accuracy as it does not connect methods called within library methods. Other tools provide more detailed information about calls outside the source of the examined project. When we make a decision about the extent to which a call graph should handle call relationships between library functions, we have to consider how important examining the dependencies generated by the execution paths is for us. In certain cases, these library functions may call the project's own methods through call back methods, creating data dependencies. Call graph based applications may be sensitive to this property. If we eliminate the library methods and their edges, the originally more detailed and less detailed call graphs will be more comparable.

Table 9 and 10 show the differences of the graphs after eliminating library calls. The graphs are more similar in most cases, but there are exceptions because we filtered out edges that were detected by both tools,

which reduced the similarity. We can observe that a lot of such edges were filtered out from Soot that were not detected by the other tools. This means that Soot represents the library nodes with more detail in the graph, which is not always necessary for every use case.

Table 11: Common calls of the Maven project between methods recognized by all tools (The number of investigated methods is 1,366).

1,567	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**1,542**	65.3%	65.37%	66.8%	59.21%	65.56%
OSA	99.6%	**1,011**	100.00%	98.91%	84.47%	99.9%
SPOON	97.49%	97.78%	**1,034**	96.71%	82.59%	97.78%
JCG	100.00%	97.09%	97.09%	**1,030**	85.44%	97.38%
WALA	100.00%	93.54%	93.54%	96.39%	**913**	93.87%
JDT	99.70%	99.61%	99.70%	98.92%	84.52%	**1,014**

Table 12: Common calls of the ArgoUML project between methods recognized by all tools (The number of investigated methods is 3,761.)

9,477	Soot	OSA	SPOON	JCG	WALA	JDT
Soot	**9,333**	44.36%	44.62%	39.8%	52.81%	42.42%
OSA	98.85%	**4,188**	99.98%	87.32%	96.35%	95.56%
SPOON	97.56%	98.10%	**4,268**	86.25%	95.10%	94.07%
JCG	99.73%	98.17%	98.82%	**3,725**	96.89%	93.32%
WALA	100.00%	81.86%	82.35%	73.22%	**4,929**	78.19%
JDT	97.8%	98.86%	99.18%	85.87%	95.21%	**4,048**

Eliminating Processing Differences. Finally, we examined the differences that come from the different processing approaches of the tools. We can see a lot of red numbers in the columns of WALA, because it ignores many nodes that are taken into account by the other tools. One possible reason can be that WALA starts the call graph builder algorithm only form certain nodes and uses the call information of the reachable nodes only. Other tools take all methods into account, while there are tools that consider only those methods that are reachable from public methods. To be able to compare the edges properly, without suffering from the differences that come from the different node sets of the graphs, we only kept those nodes (and the corresponding edges) that were found by all tools. We know that this way we lost a lot of edges, but thus we are able to compare the tools and their capabilities better and, at the same time, how the different capabilities influence the differences of the graphs. From this experiment we initially expected that we would find differences among the results of tools using different pointer analysis techniques, while tools using the same algorithms would give a much similar result. As we can see in Tables 11 and 12, the results do not support our assumption, because, for example, the results of SPOON, OSA and JDT are differing.

In order to find out the cause of the differences, we manually examined and classified them. In case of Maven, 719 out of the 1,567 edges are not found by at least one tool, which means that 46% of the edges are not "common". The ratio for ArgoUML is much worse, because 6,112 out of the 9,477 edges were not found by all tools, which is 64%. Soot or WALA apply different pointer analysis than the other tools, which explains most of the edges that were only discovered by these two tools (or only discovered by the other tools). There are 62 edges in the Maven project and 872 edges in the ArgoUML project that cause differences among the graphs, but they cannot be explained by the various pointer analysis algorithms. Their examination revealed previously undiscovered causes. Besides the reasonable differences, we found faults in the graphs as well, because the tools represented call edges that correspond to invalid call paths or execution order.

In case of SPOON and JDT, such fault was that the initializations of the static block was connected with init blocks. Besides, JDT and OSA are not able to detect the calls of class member initializations (15 edges for ArgoUML) while the other tools handle this properly. However, it is not consistent, because in some cases, JDT can recognize such edges, but OSA cannot. Another interesting observation is that SPOON inserts an extra loop edge among the init methods when the class has default constructor (19 and 42 edges for Maven and ArgoUML, respectively).

Another example for the differences is when a class A is imported from a given package but an other class A was used with a fully qualified name. SPOON did not distinguish the two classes, although only one of them was referred. Since both classes contained a method with the same signature as the invoked one, SPOON created two call edges which led to a mistake. Besides, for Maven, there were 2 cases when SPOON created call edges, although the called method had a different number of parameters than the caller provided. OSA and SPOON rarely handle overloaded[7] methods improperly, although, it may happen that a call is not connected to the correct method (1 and 2 edges for Maven and ArgoUML, respectively).

The handling of `super` classes is also not consistent among the tools. Soot, WALA and JCG connect `super.method` calls found in inner classes to the caller method in 3 cases (ArgoUML), while our JDT extension left out the `super` constructor calls 184 times for ArgoUML.

It is interesting that analyzers working on bytecode (Soot, JCG and WALA) created loops in the call graph that does not exist in the actual code (7 cases in ArgoUML). This happened when a method was overridden but the return values of the methods were dif-

[7]Method overloading allows a class to have more than one method having the same name with differing parameter lists

ferent. In this case, the compiler generates a node to handle the different return value, and the generated and the original methods are connected. Since methods cannot be distinguished based on return types, and such methods cannot be implemented, therefore, source code based analyzers did not make difference among such methods and handled them as one. This tricky solution yielded the loop edges for the bytecode analyzers, because these two "different" nodes were merged into one, and the edge between them became a loop edge.

We found a special case, when an anonymous class was implemented in a parameter list of a method and there was a call from the method of the anonymous class. Soot created a call edge from the outer method instead of the method of the anonymous class.

Finally, we already experienced in our example code that JCG handles the inherited methods in a different way which causes difference in the call graph representation. Most of the differences come from the representation, because JCG represents the inherited methods with its own node in the inherited class and the invocations refer to this generated node.

The question of RQ3 was whether, by eliminating all the known differences, we get the same graphs for the tools or not. The answer is no. As we have seen, there were so many minor differences between the graphs we did not think of as a lay user. Depending on which features are more important to us in an application (e.g.: the precision of the control flow information, or the dependencies defined between the methods), we must take into account the features of the call graph tools and choose the most appropriate for our purposes.

6 THREATS TO VALIDITY

We only collected open-source Java analyzer tools that either had an appropriate call graph output or could easily be extended with a call graph generation functionality. Although we have thoroughly investigated many other tools, we still cannot rule out the possibility of having missed some which could have fulfilled our selection requirements. In addition, the tools have many parameters that influence the construction of call graphs (e.g. different kind of pointer analysis) but since we focused on the tools instead of the difference of their algorithms, we executed each tool with only one configuration. We were not looking for an optimal setting, but one that reliably works on the tested inputs. Our goal was not to compare the call graph builder algorithms themselves, but to gather how many different outputs these tools could generate. We wanted to collect all the potential reasons for the differences of the investigated call graphs.

Moreover, in case of OSA, SPOON and JDT, we implemented the call graph exporter ourselves, therefore, it was possible for us to commit mistakes. As it was previously discussed, errors has occurred in the handling of field initializations, for example. Naturally, there may always be an error if the data extraction is left to the user.

Even though our sample code contains the features of Java 8, we did not take into account the configuration xml-s and files of the analyzed projects. The examination of runtime annotations was also ignored, because static analyzers represent the calls defined by them as calls to interface methods.

We had to develop a method name unification process to handle the different representations provided by the tools. The implemented node pairing program was tested on our sample code which contains all applicable Java 8 features. Because of the anonymous and generic elements, it was not possible to rely solely on the methods' names, we had to include the line information as well. However, the call graph tools did not always provide reliable line information, therefore, not every possible node pair was identified by the program.

7 CONCLUSION

One of the main pillars of software analysis is call graph creation, and, although, it might seem straightforward, there are countless factors that can influence the final result. Consequently, there is a wide collection of literature on call graphs. Many papers study how to improve the accuracy, completeness, or effectiveness of call graph creation, while others focus on the comparison (and, mainly, the differences) of existing approaches. Our goal was to present these differences from a practical perspective.

As the basis of our comparison, we examined how 6 open-source call graph building tools – which, while not an all-encompassing overview of the current state-of-the-art, can be considered a representative sample – perform when analyzing both an artificial example and some larger-scale, open-source projects. Before our detailed comparison, we investigated how the different calling contexts are represented by the tools. Already, in this comparison, we have seen that, in some cases, the tools also differ in their way of processing. We evaluated the impact of these differences in the resulting call graphs on 4 open source programs. The results of two (Maven and ArgoUML) were thoroughly discussed in this paper, while the

data of the rest is available in the online appendix. We tried to select the most diverse inputs possible for the analysis (e.g., libraries and executables as well). We have shown that the outputs of the different call graph creator tools may differ significantly. We have also shown on Maven and ArgoUML the extent to which the different factors affected the differences of the call graphs. With this, we have practically emphasized the parameters that can significantly determine the resulting call graphs.

ACKNOWLEDGEMENTS

Ministry of Human Capacities, Hungary grant 20391-3/2018/FEKUSTRAT is acknowledged. The project was supported by the European Union, co-fnanced by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002). This research was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things". Thanks for Tamás Aladics for the technical background work.

REFERENCES

Ali, K. and Lhoták, O. (2012). Application-only call graph construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 688–712, Berlin, Heidelberg. Springer-Verlag.

Andersen, L. O. (1994). *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen.

Antal, G., Hegedűs, P., Tóth, Z., Ferenc, R., and Gyimóthy, T. (2018). Static JavaScript Call Graphs: a Comparative Study. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM 2018, pages 177–186. IEEE.

Apache Commons (2019). Apache BCEL Home Page. https://commons.apache.org/proper/commons-bcel. [Online; accessed 2019].

Bacon, D. F. and Sweeney, P. F. (1996). Fast Static Analysis of C++ Virtual Function Calls. *SIGPLAN Not.*, 31(10):324–341.

Badenski, P. (2019). Call Hierarchy Printer GitHub Page. https://github.com/pbadenski/call-hierarchy-printer. [Online; accessed 2019].

CallGraphViewer (2019). Callgraph viewer home page. https://marketplace.eclipse.org/content/callgraph-viewer. [Online; accessed 2019].

Christodorescu, M. and Jha, S. (2003). Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA. USENIX Association.

Danny van Bruggen, Federico Tomassetti, Nicholas Smith, Cruz Maximilien (2019). JavaParser - for processing Java code Homepage. https://javaparser.org/. [Online; accessed 2019].

Dean, J., Grove, D., and Chambers, C. (1995). Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In Tokoro, M. and Pareschi, R., editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101, Berlin, Heidelberg. Springer Berlin Heidelberg.

DSE University of Szeged (2019). OpenStaticAnalyzer GitHub Page. https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer. [Online; accessed 2019].

Eclipse (2019). Eclipse home page. www.eclipse.org/eclipse/. [Online; accessed 2019].

Eclipse JDT (2019). Eclipse jdt home page. http://www.eclipse.org/jdt/. [Online; accessed 2019].

Eichinger, F., Böhm, K., and Huber, M. (2008). Mining Edge-Weighted Call Graphs to Localise Software Bugs. In *Machine Learning and Knowledge Discovery in Databases*, pages 333–348, Berlin, Heidelberg. Springer Berlin Heidelberg.

Feng, Y., Anand, S., Dillig, I., and Aiken, A. (2014). Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA. ACM.

Georgios Gousios (2019). Java Call Graph GitHub Page. https://github.com/gousiosg/java-callgraph.

Grove, D. and Chambers, C. (2001). A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746.

Grove, D., DeFouw, G., Dean, J., and Chambers, C. (1997). Call Graph Construction in Object-oriented Languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA. ACM.

Hoogendorp, H. (2010). Extraction and visual exploration of call graphs for Large Software Systems. Master's thesis, University of Groningen.

Lhoták, O. (2007). Comparing call graphs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42.

Lhoták, O. and Hendren, L. (2006). Context-Sensitive Points-to Analysis: Is It Worth It? In Mycroft, A. and Zeller, A., editors, *Compiler Construction*, pages 47–64, Berlin, Heidelberg. Springer Berlin Heidelberg.

Liu, C., Yan, X., Yu, H., Han, J., and Yu, P. S. (2005). Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. In *SDM*.

Murphy, G. C., Notkin, D., Griswold, W. G., and Lan, E. S. (1998). An Empirical Study of Static Call

Graph Extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., and Seinturier, L. (2015). Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179.

Pengő, E. and Ságodi, Z. (2019). A preparation guide for java call graph comparison: Finding a match for your methods. *Acta Cybernetica*. in press.

Reif, M., Eichberg, M., Hermann, B., Lerch, J., and Mezini, M. (2016). 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. ACM.

Reif, M., Kübler, F., Eichberg, M., and Mezini, M. (2018). Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, pages 107–112, New York, NY, USA. ACM.

Sable *J (2019). Sable *J Home Page". http://www.sable.mcgill.ca/starj/. [Online; accessed 2019].

Sable Research Group (2019). Sable/Soot GitHub Page. https://github.com/Sable/soot. [Online; accessed 2019].

Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., and Godin, C. (2000). Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345.

Tip, F. and Palsberg, J. (2000). Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, New York, NY, USA. ACM.

Wagner, T. A., Maverick, V., Graham, S. L., and Harrison, M. A. (1994). Accurate static estimators for program optimization. *SIGPLAN Not.*, 29(6):85–96.

WALA (2019). WALA Home Page. http://wala.sourceforge.net/wiki/index.php/Main_Page. [Online; accessed 2019].

Weiser, M. (1981). Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA. IEEE Press.