# Analysing and Enhancing Static Software Quality Assurance Methods

**Edit Pengő**

Department of Software Engineering
University of Szeged

Szeged, 2022

Supervisor:

Dr. István Siket

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
OF THE UNIVERSITY OF SZEGED

University of Szeged
Doctoral School of Computer Science

*"They tell me it's a suicide mission. I intend to prove them wrong."*
— Commander Shepard *(Mass Effect)*

# Preface

Ideally, in these few paragraphs, I would write about how my interest in computing has defined my life since I was young. Unfortunately, this was not the case. In the absence of quality IT education, I encountered programming relatively late, in the second year of university. But then I fell in love with it immediately. Solving programming challenges is like writing a story and thinking about the structure of the plot or the motivation of the characters. I adore both activities. That is why I decided to choose software engineering as my profession. I longed to learn its tricks at the highest possible level, therefore, I enrolled for PhD studies. I hope that throughout my many years of work, I have contributed to science and universal human knowledge.

This work could not have been finished without the help and support of my colleagues. I thank my supervisor, Dr. István Siket, for his professional help and advice. By reminding me of the deadlines, he was able to continuously motivate me even in the most hopeless situations. He could always make a comment that helped me through the challenges of research life. I would also like to thank Dr. Rudolf Ferenc, the head of the Software Engineering Department, for supporting me on my way to becoming a PhD student. I am grateful to my co-author, Zoltán Ságodi, with whom we fought our way through the difficulties of call graph comparison. It was a pleasure to explore the world of code smells shoulder to shoulder with my other co-author and partner in PhD studies, Péter Gál. I am also thankful to Dr. Judit Jász and Ervin Kóbor for the work we have done together. Many thanks to my dear friend, Edit Szűcs for her stylistic and grammatical comments on this thesis. With her humorous remarks, she was able to make even the tedious work of correcting grammar enjoyable.

Finally, I would like to express my gratitude to my parents, and my friends, András, Timi, and Adrienn, who have supported me on this hard journey. I am also grateful to my cats, Azi and Pamat for waking me up early every morning during the difficult period of working from home.

*Edit Pengő, 2022*

# Contents

# List of Tables

# List of Figures

# Listings

*To my beloved cats...*

# 1 Introduction

Developing software is a very complex task and unclear requirements, underestimated efforts, and strict deadlines make it even more difficult. Taking all these into account, it is not surprising that every software contains bugs even after its release. However, an erroneously released system can be expensive for the publisher and it can even lose its prestige and customers. Let us remember *Cyberpunk 2077* [1], the highly anticipated game of 2020. Despite the constant delays, the gaming community waited patiently for the release as it trusted the quality. Still, the game contained numerous flaws that caused extreme dissatisfaction, and sales quickly fell [7]. There have been many similar cases in recent years that have hurt the players (including the author) as well. In the case of *Mass Effect: Andromeda* [2], the negative reception due to early mistakes has resulted in this great game and magnificent series not being continued ever since.

These examples highlight how important it is for the development to be accompanied by proper and continuous quality assurance. It is a complex, multi-step process in which the goal is to reduce the impact of errors to an acceptable level. There are numerous tools and methods for quality assurance. One of the most well-known forms is software testing, which is a dynamic analysis technique. Dynamic analysis requires the execution of the software, so these methods can only be applied once executable files have been generated. In addition to dynamic analyses, there are static analysis methods. The main difference between the two techniques is that in static analysis, the test is done without running the program, so it can be applied at very early stages of development. This is advantageous because the earlier the fault is identified, the less it will cost to fix. In addition, static analyses can usually be performed at a lower cost (e.g., no need to write test cases) and can be better automated. In practice, it is worth combining the two methods of analysis, as other types of errors can usually be detected with them. The research work behind this thesis aims to help in detecting errors and coding flaws early on with static analysis techniques.

[1]Cyberpunk 2077 is an action role-playing video game developed and published by CD Projekt.

[2]Mass Effect: Andromeda is an action role-playing video game developed by BioWare and published by Electronic Arts. It is the fourth major entry in the Mass Effect series.

## 1.1 Summary of Research Results

The thesis discusses three main topics: *a heuristic-based enhancement of a Java symbolic executive engine*, *the in-depth analysis and comparison of Java static call graphs*, and *the introduction of a novel methodology for the detection of Primitive Obsession.* These topics both emphasize the importance of software quality and provide state-of-the-art solutions to combat software faults. The research on symbolic execution and Primitive Obsession is related to flaw detection itself, while the third topic in connection with call graphs goes one level deeper. Since call graphs can be extremely important in static analysis, their research supports the development of such tools.

## Part I – Heuristical Improvements of a Symbolic Execution Engine

Symbolic execution simulates the execution of a program or a part of a program using symbolic values instead of concrete ones. During regular execution, the variables of the program have concrete values, which means the program follows a specific execution path determined by these values. In contrast to that, a symbolic value is a set of concrete values. Their usage becomes interesting when a conditional statement is reached. If a logical expression contains symbolic values, the result will be symbolic too. This means that the execution can be continued on multiple paths. By default, a symbolic execution engine examines all of them, therefore, every possible execution path of a program will be explored. This way runtime failures can be detected using only static analysis information. The drawback of symbolic execution is that it requires excessive resource usage compared to normal execution. The number of execution paths can increase exponentially, for example, in the case of a symbolic cycle. Therefore, the number of explored paths is often limited, which decreases the accuracy. One possible restriction is limiting the total execution time, which leads to the omission of some of the paths. This can lead to a decrease in the accuracy of symbolic execution, e.g. not every path can be explored within a time limit, or not every part of the execution will be symbolically represented. There are several other strategies to reduce the number of paths to be explored, such as constraint solving. Constraint solving can be useful to filter out unsatisfiable paths, but calculating these can also be very resource-intensive. We introduced two heuristical methods to battle the trade-off between accuracy and resource usage. Both methods were implemented in RTEHunter [58], a Java Symbolic Execution Engine developed at the University of Szeged.

1. *Improving static initialization block handling.* The static initialization blocks of Java are executed only once when the class is loaded and initialized by the ClassLoader. They are assembled into a compiler-generated function which is called by the Java ClassLoader. We can think of these blocks as functions called automatically during the loading of the class. However, the handling of these blocks is hard to simulate in the synthetic environment of symbolic execution. When does the loading happen? It is triggered, when the class is used somehow (e.g. instance creation, usage of a static function or attribute, loaded by reflection) and has not been loaded before. As RTEHunter performs a method-by-method symbolic execution, it is difficult to precisely handle this classloading mechanism. When should RTEHunter call the static initialization blocks? Shall it call the

required static initialization blocks over and over again during the symbolic execution of each method? To overcome the difficulties of RTEHunter, we developed a filtering-based heuristical solution. It was tested on more than two hundred open-source Java systems. The results showed a notable decrease in false-positive warnings without significantly increasing the execution time or memory usage.

2. *The Null Constraint Solver.* As it was mentioned, if a conditional statement is reached during symbolic execution both the true and false execution paths will be discovered. From these conditional statements, constraints can be derived to bind the values of symbolic variables on each execution path. The logical expression that can be built from the constraints associated with symbolic variables is called the path condition. If the path condition of an execution path is infeasible, it can be pruned. However, the maintenance and the feasibility check of the path condition can be extremely resource-intensive. In this research, we applied a heuristical constraint handling mechanism called the null constraint solver. This solver tracks whether the value of a symbolic Java reference is null throughout its lifetime. This approach did not notably alter the computational time requirements of RTEHunter as no complicated feasibility check is needed. To see the improvement, we executed both the original and the modified versions of RTEHunter on 209 Java systems.

## Part II – Comparison of Static Call Graph Builder Tools

Call graphs are directed graphs representing control flow relationships among the methods of a program. Each node denotes a method, while an edge from node $a$ to node $b$ indicates that method $a$ invokes method $b$. We distinguish between two types of call graphs, static and dynamic, depending on whether they were constructed during static or dynamic analysis. Dynamic call graphs represent only those methods and call edges that were called during a specific execution of the program, therefore, they do not necessarily represent the whole program, while static call graphs attempt completeness. However, this is just an attempt, as the call graphs produced by different static analyzer tools may be considerably different. Our research on static call graphs sheds light on the many ways in which such call graphs can be constructed. As static call graphs are the main building blocks of modeling interprocedural control and data flow, it is very important to look at the factors that can cause discrepancies. The soundness of call graphs can greatly affect the results of subsequent analyses. Imagine that during symbolic execution, not the correct method is connected to the call site, but another. The whole execution path becomes useless, we have wasted execution time and resources unnecessarily. Because of this important role, we focused our research interest on the comparison of static Java call graphs.

1. *A Preparation Guide for Java Call Graph Comparison.* We collected six open-source static call graph generator tools for comparison. However, as a first step, we had to make the graphs comparable. Methods developed for general-purpose graphs cannot be directly applied to call graphs. Even if the structure of two call graphs is isomorphic, they can be considered completely different because of the labeling of the nodes. Therefore, we developed a multi-step solution to find

a mapping between their nodes. The basis of the pairing is the methods' fully qualified name. However, we had to add heuristic as well in order to handle, for example, anonymous and generic source code elements. Our heuristical steps have improved the basic pairing by 2-3 percent for some tools. This is a considerable enhancement, however, a significant number of nodes remained unmatched. We manually investigated the root causes for this. Our thorough examination revealed that most of the unmatchings cannot be resolved. These are due to the differences in the operation of the tools.

2. *Systematic Comparison of Six Open-Source Java Call Graph Construction Tools.* After developing the pairing mechanism, we performed an in-depth analysis of the remaining differences. We challenged the six tools on an example code containing the language features of Java 8 and on multiple real-life open-source Java systems and performed a quantitative and qualitative assessment of the resulting graphs. We determined the factors that cause discrepancies in the graphs: the handling of initializer methods, polymorphism, Java 8 language elements (such as lambdas), dynamic method calls, generic source code elements, and anonymous source code elements. The different treatment of these features can cause significant differences in the generated graphs. This research highlighted how challenging the richness of the Java language can be in generating call graphs. These differences have to be taken into consideration when developing a call graph-based tool.

## Part III – Development and Evaluation of Primitive Obsession Metrics

Primitive Obsession is a code smell that can be vaguely interpreted as the overuse of primitive data types. As a code smell, it can be a useful indicator of underlying design problems and might suggest the need for refactoring in the code. Despite its potential benefits, it has not received significant attention from researchers. Therefore, we decided to study Primitive Obsession and defined multiple metrics that can highlight the potential places where this bad smell could occur.

1. *An evaluation of the Primitive Obsession metrics.* The smell can take many different forms, therefore, it was necessary to define several indicators. I introduced three metrics to describe two important aspects of Primitive Obsession. If a class suffers from Primitive Obsession, certain method parameters will appear multiple times in the parameter lists of its methods. The first metric grasps this symptom, while the other two metrics are for type code[3] detection. The metrics were integrated into the OpenStaticAnalyzer (OSA) [5] static analyzer tool and they were evaluated on three real-life Java systems and a small, Primitive Obsession-infested project. The manual examination of the results has shown that the warnings draw attention to classes and methods in which primitive types are used too carelessly.

2. *Examining the Bug Prediction Capabilities of Primitive Obsession Metrics.* In this research, we expanded an existing bug prediction database [37] with the three Primitive Obsession metrics and examined the effect on the predictive ability. By

[3]Type codes are primitve types that simulate types, e.g. different types of vehicles

learning from the bugs of the past and source code features it builds a prediction model to foretell the location and amount of future bugs. We studied to what extent the metrics contribute to the prediction ability of the original dataset. First, we evaluated the original and the extended bug datasets with the J48 algorithm, then performed a cross-project validation, meaning that we trained a model on each project and evaluated it on every other project. The overall results showed that the metrics could improve the prediction in certain cases. We also demonstrated with Principal Component Analysis (PCA) [89] that metrics contribute to the variance of the extended data set, meaning that they provide a new and useful approach to describe and measure the source code.

## 1.2 Structure of the Dissertation

The thesis consists of three main parts, which are also the three thesis points. Each part consists of four chapters. The first chapter provides an introduction into the research area, the next two chapters discuss the constributions of the thesis, organised by the articles on which they are based. The last chapter in each part summarizes the results. The mapping of the chapters and the corresponding thesis points is illustrated in Table 1.1.

| **№** | **Thesis points** | **Chapter** |
|---|---|---|
| I. | Heuristical Improvements of a Symbolic Execution Engine | 2 - 5 |
| II. | Comparison of Static Call Graph Builder Tools | 6 - 9 |
| III. | Development and Evaluation of Primitive Obsession Metrics | 10 - 13 |

**Table 1.1:** Mapping of thesis points and chapters

The **first part**, which describes our research in symbolic execution, includes 4 chapters. Chapter 2 briefly introduces the theory of symbolic execution and currently used technologies. In Chapter 3, we explain the heuristical handling of static initializer blocks and present the results. Chapter 4 discusses our research on a heuristical constraint solving approach. Chapter 5 summarises the thesis point.

The **second part** deals with the comparison of Java static call graphs. This work is also divided into four chapters. Chapter 6 provides background and enumerates the static analyzer tools we used in the research. Chapter 7 contains the first main phase of our work, the unification process during which we made the call graphs comparable. The actual comparison and its conclusions are presented in Chapter 8. The thesis point is summarized in Chapter 9.

Finally, in the **third part**, the research on Primitive Obsession is discussed. Primitive Obsession and the metrics that we designed for its detection are described in Chapter 10. The first evaluation of these metrics on real-sized Java systems and the results are summarized in Chapter 11. This research was continued by examining the bug prediction capabilities of the Primitive Obsession metrics, which are discussed in Chapter 12. The summary of the thesis point is given in Chapter 13.

Chapter 14 sums up the thesis, while in appendices A and B, brief summaries of the thesis are shown in English and Hungarian, respectively. The appendices, furthermore, contain the thesis points, as well as the author's contributions, and the underlying publications.

# Part I

# Heuristical Improvements of a Symbolic Execution Engine

# 2 Theory of Symbolic Execution

## 2.1 Overview

Testing is a critical part of software development for identifying bugs and runtime issues. The goal is to detect bugs in development as soon as possible, thus reducing the cost of fixing them. The testing process is highly dependent on the development methodology used during development. Some methodologies, such as Extreme Programming, place great emphasis on creating software tests before implementation. However, in the case of the waterfall model, the testing process only begins after the implementation phase. Another important factor is that sophisticated, high-coverage testing requires a significant amount of resources which is not always available during the pressure of production. Therefore, it is an excellent addition to traditional (dynamic) testing if we introduce static analysis as an automated method that supports bug detection.

Static analyzer tools help finding issues by analyzing the source code (and, optionally the documentation) itself. Modern IDEs contain such tools to aid quality software development, and many open source and commercial static analyzer programs are available as well. This way, the developer receives feedback even if the project is not compilable yet and certain bugs can be detected immediately. Static analyzers can be faster and easier to use than executing an entire test suite especially if it contains manual tests. They help adhere to strict development standards, which reduces potential production issues.

Static analyzers detect other types of defects compared to dynamic testing techniques. The time and computational effort required for the analysis depend on the complexity of the aimed issues. IDE integrated and open-source tools usually perform the analysis quick and detect problems like dead code, whilst for serious problems a more time-consuming deep static analysis is needed. Simpler analyzer tools include, for example, rule checkers and linters. Linters can detect if a method is longer than a given number of lines, which is a type of code smell. Compared to these undemanding techniques, the classical symbolic execution performs a complex analysis by simulating the execution of the program with the use of static analysis information only. This way, it is possible to find more severe runtime issues that might have remained undiscovered even after manual testing. We will see in later chapters that symbolic execution is slow and resource-intensive, so trade-offs need to be made for usable results.

This chapter introduces the theory behind symbolic execution. Section 2.2 presents the theoretical background through an example while Section 2.3 introduces the symbolic execution engine used for this research. Section 2.4 enumerates the most important research papers dealing with symbolic execution and related subjects.

## 2.2 Background

During regular execution, the variables of the program have concrete values, meaning that the program follows a specific executional path determined by these values. The basic idea of symbolic execution is that the program is executed on symbolic values instead of concrete values. This execution cannot be considered a real execution, it only means an execution within a synthetic environment, by a static analyzer. When the exact value of a variable cannot be decided (because, for example, it is a user input or method parameter), a symbolic value is assigned to it. A symbolic variable can contain multiple concrete values that are allowed for its type, for example, a symbolic integer can take arbitrary values within the range of integer type.

The possible values of symbolic variables can be bound with constraints. These constraints are usually derived from conditional statements or assignments. If a statement contains symbolic variables, the whole statement will be symbolic, and this applies for logical expressions as well. A symbolic boolean can either be true or false, therefore, the symbolic engine will continue to execute both the true and false branches of a symbolic conditional statement. This means that theoretically every possible executional path will be explored, and hidden runtime exceptions can be detected.

The tree built up from the executional paths is called the symbolic execution tree (an example is presented in Figure 2.1). This is a directed, acyclic graph where each node represents a state of the program. It is clear that some branches of the symbolic execution tree are unreachable even with symbolic values because the concatenation of the conditional statements leading to them are unsatisfiable. Symbolic engines can cut off these unnecessary branches by maintaining and checking the satisfiability of a path condition (PC). The path condition is a quantifier-free logical formula over the symbolic variables. It contains constraints derived from the conditional statements that have to be satisfied in order to reach the investigated executional state. Symbolic engines use constraint solver algorithms to check the satisfiability of the path condition and to decrease the number of possible branches when a conditional statement is reached. On the contrary, if a path condition is satisfiable, assigning the solutions to the symbolic variables as input values will direct the concrete execution to the state of that PC. Therefore, for example, symbolic execution can be a tool for test input generation.

```java
public static int isTriangleValid(Point p1,Point p2,Point p3){

  double a = euclideanDistance(p1,p2);
  double b = euclideanDistance(p2,p3);
  double c = euclideanDistance(p1,p3);

  if(a == b+c || b == a+c || c == a+b) {
    return -1;
  }
10
11    if(a > b+c || b > a+c || c > a+b) {
12      return 1;
13    }
14    return 0;
15  }
```

**Listing 2.1:** Sample code that checks if three points can form a triangle

Listing 2.1 shows an example function which decides if the three 2-dimensional points given as parameters can form a triangle. There are two erroneous return values: the function returns with -1 if the three points are on a line, and with 1 if the three points are not on one line but the triangle inequality is not satisfied anyway. The return value is 0 if a proper triangle can be composed from the three points. The implementation of the `euclideanDistance` function is not present, but as its name suggests it calculates the distance between two 2-dimensional points with the classic euclidean formula. Suppose that the symbolic execution was started with this function, so the actual value of parameter `p1`, `p2`, and `p3` are unknown, therefore, they must be handled symbolically. This means that the two data members (coordinate `x` and `y`) of each `Point` are symbolic too, so they can hold any value from the domain of their type. Therefore, the value returned by the `euclideanDistance` function can be symbolic, making local variable `a`, `b`, and `c` also symbolic variables. Theoretically, we have more information about these variables than the parameters of `isTriangleValid` because the symbolic engine executes their initialization. Nonlinear constraints can be derived from the euclidean formula and then appended to the path condition, however, it is clear that doing it programmatically is not a trivial task. Figure 2.1 shows the symbolic execution tree built up during the execution of the sample code. Each conditional statement at line 5 and 8 can be split up to three sub-expressions, all of them containing symbolic variables. The short circuit evaluation gives an explanation as to why the two `if` statements create three-three branching points during the execution, whilst having symbolic variables in the conditions means that both the true and false outcomes of each sub-expression have to be investigated.

Path conditions are not presented in Figure 2.1 because - as previously mentioned - the constraints derived from the initialization are non-linear. We assume that symbolic engines will not build up and solve constraints like that on their own, mostly because they treat the `sqrt` function grammatically with symbolic variables and an indefinite number of loops instead of using its mathematical meaning. However, a developer is able to detect straightaway the three unreachable states, namely the three `return 1` branches (colored with yellow). If three points given on the two dimensional plane are not arranged on a line they definitely form a triangle. Therefore, the `if` condition in line 8 is unnecessary, mathematically infeasible, making the true branches of each sub-expression unreachable. If an unreachable program state is discovered through unsatisfiable path conditions, the state and the sub-tree derived from it should be pruned from the symbolic execution tree. Not only do we avoid needless computations on that path, but possible false-positive warnings will also be eliminated.

The example in Listing 2.1 gives a quick insight into how the symbolic execution is performed in the classical way. Whilst it is a very powerful tool for detecting runtime failures, it has several limitations. One of them that already occurred in the sample code is related to constraint solving. As the example shows, it is not a trivial task to find an efficient solution to reveal unreachable program states. Another major

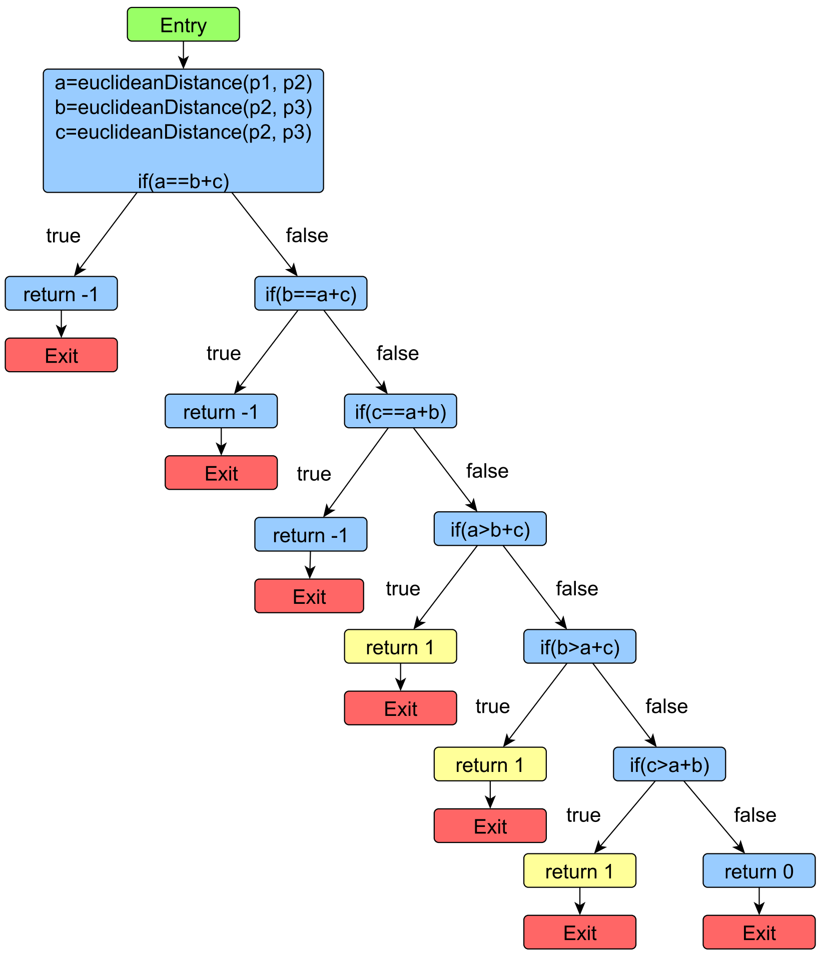

**Figure 2.1:** Symbolic execution tree constructed from `isTriangleValid` (see Listing 2.1)

drawback is path explosion. The number of possible executional paths grows with the number of conditional statements almost exponentially. The scenario is even worse with switch statements and loops containing symbolic conditions. Since it is not known in advance how many iterations have to be done in a loop, symbolic engines have to make a guess or unroll the cycle until a limitation is reached, otherwise an infinite number of branches is created. Managing different kinds of features in programming languages symbolically, like static initialization in Java, also gives a room for challenges. Simulating the environment of a program (e.g. system calls) in a purely symbolic way can also be a problem. For a more detailed summarization of symbolic execution and its difficulties see the survey of Baldoni et al. [16].

## 2.3 RTEHunter

RTEHunter (abbreviation of RunTimeException Hunter) is a Java symbolic execution engine. It is part of the SourceMeter project, which is developed at the Department of Software Engineering, University of Szeged. SourceMeter [91] analyzes C/C++, Java, C#, Python, and RPG projects. It calculates source code metrics, detects code clones, and finds coding rule violations in the source code. RTEHunter is one of the static analyzers of the SourceMeter Java toolchain. It is designed to detect runtime exceptions in Java source code without actually executing the application in a real-life environment. Currently it can detect four kinds of common failures: *NullPointerException*, *ArrayIndexOutOfBoundsException*, *NegativeArraySizeException*, and *DivideByZeroException*.

RTEHunter performs the analysis by calling the symbolic execution for each method in the program separately. For big systems, this approach is usually a better solution than only starting the execution from the `main()` method [57, 81]. Starting the execution at the entry point of the real-life execution seems like a natural and convenient idea, however it is clear that the practical limitations mentioned in the previous section would be reached very soon, leaving many parts of the code unexplored. RTEHunter limits the number of states (nodes) in the symbolic execution tree and the depth of the execution tree as well.

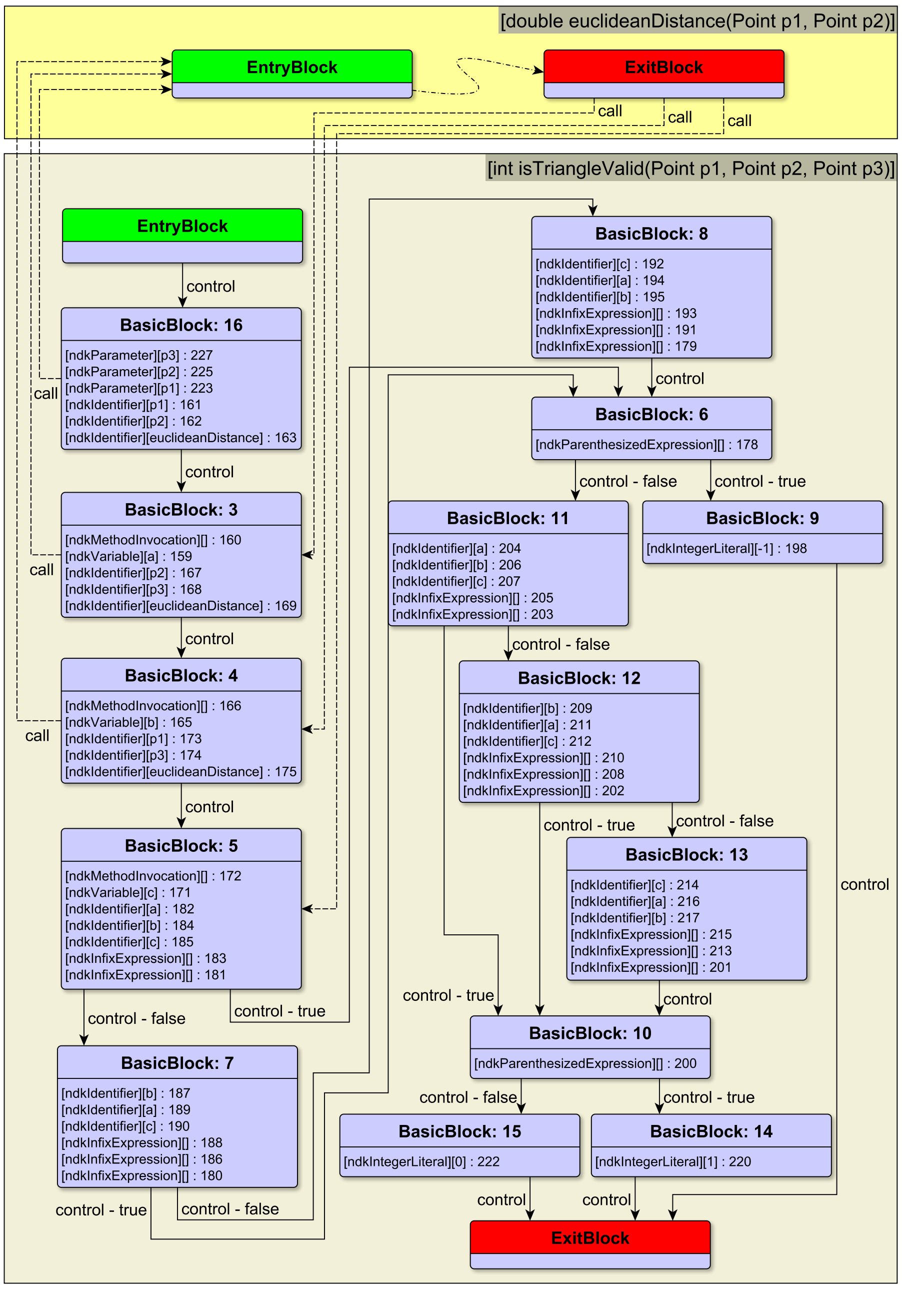
[double euclideanDistance(Point p1, Point p2)]
EntryBlock
ExitBlock
call
call
call
[int isTriangleValid(Point p1, Point p2, Point p3)]
EntryBlock
control
BasicBlock: 16
[ndkParameter][p3] : 227
[ndkParameter][p2] : 225
[ndkParameter][p1] : 223
[ndkIdentifier][p1] : 161
[ndkIdentifier][p2] : 162
[ndkIdentifier][euclideanDistance] : 163
call
control
BasicBlock: 3
[ndkMethodInvocation][] : 160
[ndkVariable][a] : 159
[ndkIdentifier][p2] : 167
[ndkIdentifier][p3] : 168
[ndkIdentifier][euclideanDistance] : 169
call
control
BasicBlock: 4
[ndkMethodInvocation][] : 166
[ndkVariable][b] : 165
[ndkIdentifier][p1] : 173
[ndkIdentifier][p3] : 174
[ndkIdentifier][euclideanDistance] : 175
call
control
BasicBlock: 5
[ndkMethodInvocation][] : 172
[ndkVariable][c] : 171
[ndkIdentifier][a] : 182
[ndkIdentifier][b] : 184
[ndkIdentifier][c] : 185
[ndkInfixExpression][] : 183
[ndkInfixExpression][] : 181
control - false
control - true
BasicBlock: 7
[ndkIdentifier][b] : 187
[ndkIdentifier][a] : 189
[ndkIdentifier][c] : 190
[ndkInfixExpression][] : 188
[ndkInfixExpression][] : 186
[ndkInfixExpression][] : 180
control - true
control - false
BasicBlock: 8
[ndkIdentifier][c] : 192
[ndkIdentifier][a] : 194
[ndkIdentifier][b] : 195
[ndkInfixExpression][] : 193
[ndkInfixExpression][] : 191
[ndkInfixExpression][] : 179
control
BasicBlock: 6
[ndkParenthesizedExpression][] : 178
control - false
control - true
BasicBlock: 11
[ndkIdentifier][a] : 204
[ndkIdentifier][b] : 206
[ndkIdentifier][c] : 207
[ndkInfixExpression][] : 205
[ndkInfixExpression][] : 203
BasicBlock: 9
[ndkIntegerLiteral][-1] : 198
control - false
BasicBlock: 12
[ndkIdentifier][b] : 209
[ndkIdentifier][a] : 211
[ndkIdentifier][c] : 212
[ndkInfixExpression][] : 210
[ndkInfixExpression][] : 208
[ndkInfixExpression][] : 202
control - true
control - false
BasicBlock: 13
[ndkIdentifier][c] : 214
[ndkIdentifier][a] : 216
[ndkIdentifier][b] : 217
[ndkInfixExpression][] : 215
[ndkInfixExpression][] : 213
[ndkInfixExpression][] : 201
control
control - true
control
BasicBlock: 10
[ndkParenthesizedExpression][] : 200
control - false
control - true
BasicBlock: 15
[ndkIntegerLiteral][0] : 222
BasicBlock: 14
[ndkIntegerLiteral][1] : 220
control
control
ExitBlock

**Figure 2.2:** Control flow graph of `isTriangleValid` (see Listing 2.1)

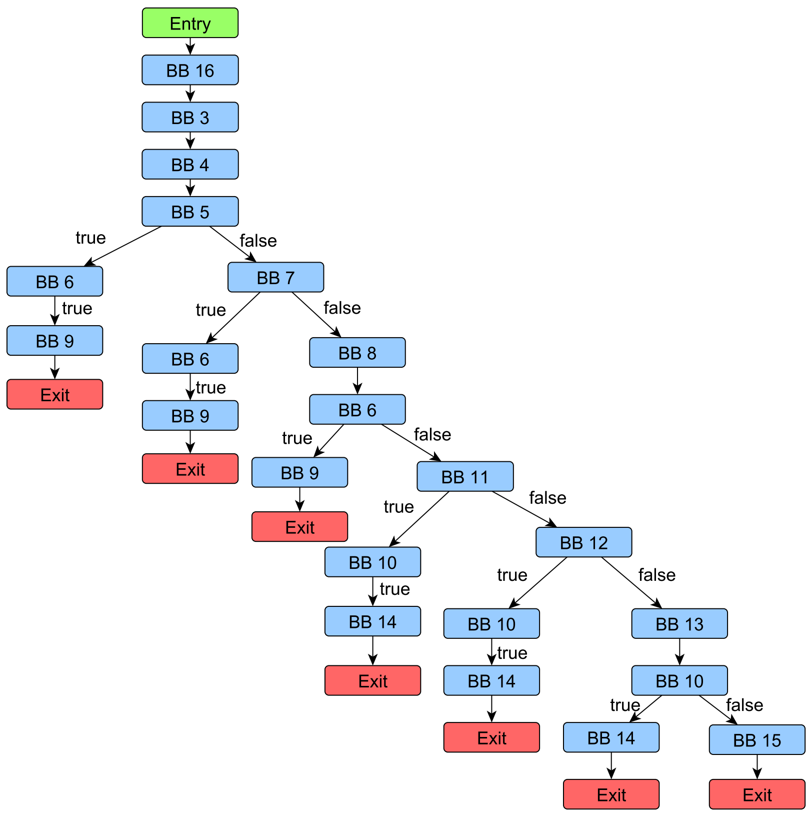

**Figure 2.3:** Symbolic execution tree of `isTriangleValid` built by RTEHunter (see Listing 2.1)

RTEHunter uses the Abstract Semantic Graph (ASG) [36] of the program, which is constructed by the analyzer of SourceMeter. The ASG is a language-dependent representation of the source code that contains every detail of the source code in an internal graph representation. It is similar to an Abstract Syntax Tree, but provides additional semantic information. First, RTEHunter builds a language-independent Control Flow Graph (CFG) [11] based on the ASG, and symbolic execution works on this CFG. In a CFG, each node represents a *basic block*. Basic blocks are the abstraction of straight-line program parts which are guaranteed to be executed sequentially. A jump in the code, like a conditional statement or a return statement, terminates the current basic block, and the outgoing edges connect to the basic blocks that are the targets of that particular jump. BasicBlock 5 in Figure 2.2 shows an example for a conditional jump with a `true` and `false` outgoing edge. Since RTEHunter performs symbolic execution on each method of the program, a separate control flow graph will be created for all methods. Each CFG for a method contains exactly one entry block and exactly one exit block, even if there are more `return` statements in a method (see Figure 2.2). The control can only enter and leave the CFG of a method through these two blocks, making them useful for handling function calls and returns.

Figure 2.2 shows the CFG built up by RTEHunter for the `isTriangleValid` function defined in Listing 2.1. The `euclideanDistance` function has a different control flow graph shown on the top of the figure. It is reached through call edges, however, this CFG is only represented with the entry and exit blocks without real content. Inside each basic block, we can find the ASG nodes that are visited in a sequential order. As previously mentioned, each method invocation creates a new basic block, which is why the first three initialization lines of the `isTriangleValid` method appear in four basic blocks (BB 16, 3, 4 and 5). From BasicBlock 5, a new basic block starts at each conditional sub-expression. Note that BasicBlock 6 represents the whole conditional statement at line 5, whilst BasicBlock 10 represents the `if` statement of line 8. BB 6 and BB 10 are not necessary, they are collector basic blocks introduced to make handling the expressions that contain multiple sub-expressions easier. Naturally, there are also `true` and `false` out-edges of these basic blocks, but the RTEHunter is aware of the short circuit evaluation so, for example, if the control reaches BB 6 from the `true` edge of BB 5, there will be no branching in BB 6, only the `true` edge will be continued. Figure 2.3 shows the symbolic execution tree constructed by RTEHunter. Some basic blocks appear multiple times, but each of them represents a different program state.

It can be seen from the given control flow graph that RTEHunter performs an interprocedural analysis. The symbolic engine handles method calls by continuing the symbolic execution in the called methods and then returning to the original function as during normal execution. Built-in or third-party Java functions are also executed, however any warning found in them will be filtered out from the final output.

## 2.4 Related Work

The idea of symbolic execution was introduced in the 1970s as an elegant and powerful method for software proving, validation, and test generation. In 1976, King introduced the fundamentals of symbolic execution along with the presentation of the EFFIGY system [61]. EFFIGY is one of the first symbolic executor engines. It is written for PL/I programs and only handles integer variables symbolically. With its interactive user interface, EFFIGY lets programmers explore the symbolic execution tree for exhaustive software testing. Another important early work was the SELECT system for the systematical debugging of LISP programs. SELECT was published by Boyer et al. in 1975 [21]. Like EFFIGY, it aims at systematical and exhaustive testing, it allows verifications by user-supported asserts, and it also provides interactivity in the control of symbolic execution.

Another important early work is the survey of Coward [29] from 1988. Coward collected and analyzed six symbolic executor systems containing the previously mentioned EFFIGY and SELECT systems as well. He compared the investigated systems to an ideal symbolic executor engine revealing the major limitations and challenges, like the handling of path explosion and constraint solving that still have to be faced nowadays. The survey also provided a classification for symbolic engines according to their main purpose: test data generation from constraints applied to symbolic variables, program proving with the use of assertions, program reduction, path domain checking, and symbolic debugging for tracing runtime issues.

Later, to overcome the challenges, it became the new trend that the symbolic engines did not use pure symbolic execution, instead, they relied on hybrid techniques to overcome the limitations of symbolic execution. The mixture of concrete and symbolic execution is called concolic execution, or in other words, dynamic symbolic execution. The DART (Directed Automate Random Testing) system introduced by Godefroid et al. [45] was the first to suggest this approach. It performs a directed search in the symbolic execution tree through the iterated executions of the investigated software. The process starts with random input values, then during each execution information is collected about the symbolic variables to produce input data that would force the next execution on a new path. CUTE and jCUTE are also concolic testing systems for C and Java [87, 69]. Similar to the DART system, they collect information about symbolic constraints to produce input data, but they are extended with the capability of handling multithreading. If the constraint solver fails to satisfy a complex expression, the constraints are simplified by replacing some symbolic variables with concrete values. jCUTE can generate JUnit tests for sequential software. For .NET programs, Pex [95] is used for generating test suites with the help of dynamic symbolic execution. It can be integrated into Visual Studio as an add-in. Anxiety is another dynamic symbolic execution engine from recent years, developed at the Ivannikov Institute for System Programming of the Russian Academy of Sciences[44]. SAGE (Scalable, Automated, Guided Execution) [46] is a whole-program whitebox fuzzing program for x86 binaries. It employs a new algorithm called generational search that addresses the aforementioned limitations of symbolic execution: path explosion and imprecision (e.g. imprecise handling of system calls). The generational search-based concolic execution algorithm has a novel, open-source implementation as well, the RiverConc concolic engine [76]. It uses reinforcement learning to optimize the number of SMT queries. RiverConc is also developed for detecting defects in x86 binaries.

EXE (EXecution generated Executions) [23] and its successor, KLEE [22] are better known symbolic executors of recent times. They symbolically execute C programs trying to explore all possible paths. When an error is reached, the symbolic constraints are used for test input generation. KLEE is based on the LLVM assembly language and functions as a virtual machine. It powerfully handles environmentally-intensive programs, for example by setting up a symbolic filesystem or by simulating faulty system calls. Because it is such an effective tool, KLEE is the basis for countless studies on symbolic implementation. For example, there is a novel regression learning-based search strategy called LEARCH [53] which was implemented on KLEE. LEARCH effectively explores and selects promising states for symbolic execution, thus addressing the path explosion problem. It uses an iterative learning process using the data extracted from the symbolic execution of the program and generates high-quality, diverse tests.

Similar to KLEE, the Java PathFinder (JPF) [4] tool also behaves like a virtual machine and executes the Java bytecode in a special way. JPF is developed at the NASA Ames Research Center for verifying and checking NASA projects. It has an extension, called Symbolic PathFinder (SPF) [79], for performing symbolic execution. SPF supports concolic execution and can be customized with several constraint solvers. One of them is the CORAL solver [92], which handles complex mathematical functions, making it effective in scientific domains.

Lukow et al. created a novel dynamic symbolic execution engine called JDART built on top of the Java PathFinder [67]. Its goal is to handle industrial scale, complex software including NASA systems. JDART executes the Java bytecode and performs dynamic symbolic analysis of the specified methods. It can generate JUnit test suites that exercise all the program paths found by JDART. JDART was extended by Mues and Hower in 2020 [74] for the SV-COMP2020 competition [18].

SymJEx is a novel symbolic engine from 2020 based on the GraalVM [62]. Currently, it analyzes Java programs, however, as the virtual machine supports multiple languages, it can be expanded. SymJEx detects errors and generates test cases for them.

Kádár et al. created a wrapper for the previously mentioned SPF. It is called JPF Checker [57]. SPF can perform symbolic execution starting only from the `main` method, therefore, JPF Checker generates classes with a `main` method for each function in the investigated software, therefore, it executes the whole code base. Thus, it does not only run the methods that are available from the `main` method.

JPF Checker performs the execution in the style of RTEHunter, but with the use of SPF as a symbolic engine. The predecessor of RTEHunter, named Symbolic Checker, was introduced by Kádár et al. in 2015 as a subject for improved constraint solving technique [58]. The performance of the two symbolic engines was compared in this work.

Two relatively recent surveys on the subject of Symbolic Execution are the work of Cadar and Sen [24] which was presented in 2013, and the detailed paper of Baldoni et al. from 2016 [16]. Both give an overview of the state of the art, the challenges, and their current solutions.

# 3 Improving Static Initialization Block Handling

## 3.1 Overview

A symbolic execution engine has to overcome many practical limitations to be able to produce useful results efficiently. Some behaviors are hard to simulate in the synthetic environment of symbolic execution. The handling of the static initialization blocks in Java is a good example. RTEHunter ignored these blocks, therefore, we decided to improve its execution. We first looked at how static initialization blocks work in Java and then worked out a solution that gives enough accuracy but does not significantly increase runtime. We described this research in our paper published in 2017 [118].

The rest of this chapter is structured as follows. In this chapter, we start by giving a short insight into the difficulties of handling the static initialization blocks in Section 3.2, then Section 3.3 describes our solution that was implemented in RTEHunter. The results are discussed in Section 4.4.

## 3.2 Static Intialization Blocks

A static initializer block is a `static{}` block of code in a Java class. It is executed only once when the class is loaded and initialized by the ClassLoader. A class can have multiple static initialization blocks which can appear anywhere inside the class body and are executed in the order of appearance. Typically, their purpose is the initialization of static class members. Static blocks and initializers are assembled into a compiler generated function. Therefore, we can think about these blocks as a function called automatically during loading the class.

The behavior of the Java ClassLoader is well defined; a class is loaded only when certain circumstances are met: the class is used somehow (e.g. instance creation, usage of a static function or attribute, loaded by reflection) and has not been loaded before. For a static analyzer tool like RTEHunter that does not actually execute the Java bytecode, it is difficult to handle the Java classloading mechanism. The evident solution would be to implement a module similar to the Java ClassLoader itself. This module would produce a CFG composed from the ASG nodes of the static initializer blocks found in a class, then execute it when the engine meets a class for the first time. If we consider that RTEHunter performs a method-by-method execution, it is clear that each class has to be loaded every time when the symbolic engine starts to execute a new function. With this approach, we add not only extra function calls, but execute the initializer method for a class repeatedly. Unfortunately, even with this solution it is not guaranteed that the values of the static variables are correct because in real execution they can be changed, since they were given initial values at the time when their classes were loaded.

Listing 3.1 and Listing 3.2 show a very simplified example for the static initialization of two classes. Class `A` uses the static member of class `B`, therefore, there is a dependency between the two static blocks. The sample code also suggests that loading a class and executing its constructed static initializer function might cause the recursive loading of other classes (String and Integer in the example). This recursion may add so many extra nodes to the symbolic execution tree that it can easily reach its limitation before providing any useful result about the examined method.

```java
class A {
  public static String STR;

  static {
    STR = B.NM.toString();
  }
}
```

**Listing 3.1:** Example class A

```java
class B {
  public static Integer NM;

  static {
    NM = new Integer(6);
  }
}
```

**Listing 3.2:** Example class B

We concluded that this closer-to-reality approach is very powerful and helps with detecting runtime issues in connection with static initialization, however, it gives an overhead to the symbolic execution. The practical, limits like the number of states or the depth of symbolic execution tree, would be reached sooner, especially in complex methods, leaving the interpreation of the function body unfinished. A natural idea for improving the classloader solution is to run each static initializer block only once before the symbolic execution of methods starts. This technique lessens the overhead and the computational time, but does not solve several other considerable problems. A dependency graph is needed for setting up the proper initialization order. Methods can change the value of static member variables making them invalid for other methods, so the symbolic engine should be able to reset the initial values for these fields after each method-execution. This means extra-memory usage and the cautious handling of static class members.

## 3.3 Contribution

During the investigation of the two solutions listed in the previous section, we found out that integrating one of them to the RTEHunter would provide a system whose overall behavior was still not close enough to real life. For an engine which starts the execution from the `main` method, like the Symbolic PathFinder [79] does, the effort for creating a ClassLoader module might be rewarding. However, for RTEHunter the trade-off between the complexity and the usefulness of such a module is not so promising, therefore, we decided to choose a simpler but still fast and usable approach. Since it starts a symbolic execution from each method, it is not trivial to interpret how a ClassLoader would work realistically.

The previous version of RTEHunter neglected the handling of static initializer blocks. Examining the reported warnings showed that this shortage caused many false-positive errors. Most of them were `NullPointerExceptions` originated from uninitalized static class members. We decided to work out a solution that would not alter the current computational time and effort needs of RTEHunter very much, but would still be capable of eliminating the false-positive warnings. From a user perspective, it is more important to detect fewer but more precise runtime faliures.

The result of our idea was a filtering mechanism. Instead of mimicing the behavior of Java ClassLoader, we simply checked when a `NullPointerException` arose, if the variable associated with this warning was initialized in the static initializer blocks of its class. If the answer was positive, the warning was treated as a false-positive.

## 3.4 Results

We collected 209 open source Java systems to test our solution. These Java systems were various; they were from different domains and their size ranged from a thousand to 2.5 million lines of code. Table 3.1 shows the minimum, maximum, and the average lines of code (LOC) and the number of classes (NOC) metric of the systems. The histogram of the LOC is presented on Figure 3.1.

| | Lines of Code | Number of Classes |
|---|---:|---:|
| Minimum | 993 | 17 |
| Maximum | 2,594,569 | 16,934 |
| Average | ∼ 140,653 | ∼ 1,090 |

**Table 3.1:** The Lines of Code and Number of Classes metrics of the examined Java projects

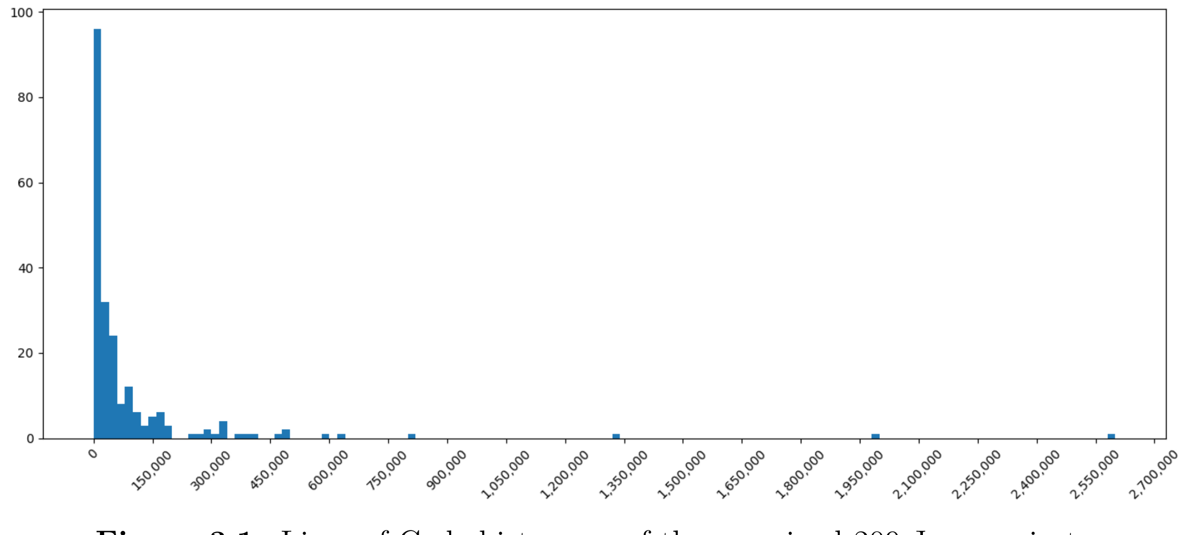

**Figure 3.1:** Lines of Code histogram of the examined 209 Java projects

To see the improvement of our approach, we executed both the original and the improved versions of RTEHunter on the Java systems. The results are presented in Table 3.2. As it can be seen, the number of NullPointerException warnings (NPE) decreased by 237, the number of ArrayIndexOutOfBoundsException warnings increased by 2, while the other two remained the same. We compared the warnings and found that in fact 242 NPE warnings were eliminated and 5 new NPE warnings appeared in the results of the improved RTEHunter. RTEHunter stops the execution along a path when a runtime failure is detected, therefore, it is obvious that the 7 new warnings showed up because our filtering mechanism let the engine go deeper in the symbolic execution tree. 10% of the disappeared runtime failures were manually investigated to check whether they were really false-positives. The conclusion of the verification was that not only did we eliminate more than two hundred erroneous NPE warnings, but also discovered 7 true positive runtime issues, which is about 7% improvement.

| **Warning type** | **Original RTEHunter** | **Improved RTEHunter** |
|---|---|---|
| NullPointerE. | 3,369 | 3,132 |
| NegativeArraySizeE. | 24 | 24 |
| DivideByZeroE. | 167 | 167 |
| ArrayIndexOutOfBoundsE. | 679 | 681 |

**Table 3.2:** The results of the original and improved RTEHunter executions

## 3.4.1 Example

An example of the eliminated false-positive errors is presented in Listing 3.3 and Listing 3.4. This simple example was taken from the Spring-Framework[1] keeping only the necessary lines.

Listing 3.3 shows a portion of the `JafMediaTypeFactory` class of the Spring-Framework. It has a static class member called `fileTypeMap`, that is initialized in a static initialization block. Listing 3.4 introduce the corresponding code part from the `FileTypeMap` class of the `javax.activation` package. Although it is clear at a glance that the `fileTypeMap` class member cannot take a null value, a NullPointerException is reported on Line 9 of Listing 3.3. RTEHunter executed the `initFileTypeMap()` method independently of the static initializer block, but it did not use its return value to initialize the `fileTypeMap` class member, as the static initializer block itself was not executed.

Thanks to our heuristic solution, these types of false-positive errors have been fixed despite not modeling a complete ClassLoader module. The solution has become part of the SourceMeter system.

```
1  private static class JafMediaTypeFactory {
2    private static final FileTypeMap fileTypeMap;
3
4    static {
5      fileTypeMap = initFileTypeMap();
6    }
7
8    public static MediaType getMediaType(String filename) {
```

[1]`https://github.com/spring-projects/spring-framework`

```
    //NPE:
    String mediaType = fileTypeMap.getContentType(filename);
    return (StringUtils.hasText(mediaType) ? MediaType.↩
        parseMediaType(mediaType) : null);
  }

  private static FileTypeMap initFileTypeMap() {
    //...
    return FileTypeMap.getDefaultFileTypeMap();
  }
}
```

**Listing 3.3:** The problematic method and the static initialization block of the `JafMediaTypeFactory` class

```
public abstract class FileTypeMap {

  public static FileTypeMap getDefaultFileTypeMap() {
    if (defaultMap == null)
        defaultMap = new MimetypesFileTypeMap();
    return defaultMap;
    }
}
```

**Listing 3.4:** The called method of the `javax.activation.FileTypeMap` class

## 3.5 Conclusion

With this research, our aim was to enhance an already operating symbolic execution system, the RTEHunter. We developed a heuristic solution for executing static initialization blocks. These blocks are only executed once when the class is loaded to initialize the static class members. This may seem insignificant at first but the proper initialization of variables is extremely important for the precise symbolic execution. Since RTEhunter had not handled these blocks before, it resulted in a lot of false-positive errors. These false-positive errors also led to a halt the execution although it could have continued and thus other errors remained hidden. Because of the method-by-method operation of RTEHunter, we did not consider the proper, symbolic implementation of the class loader, instead we applied a heuristical solution. With this heuristic, in case of an NPE of a static class member, we checked whether the variable was initialized in its class's static initializer block and if the answer was yes we dissmised the exception. We tested this solution on numerous real-sized Java systems and manually verified the results. The conclusion of the verification was that not only did we eliminate more than two hundred erroneous NPE warnings, but also discovered 7 true positive runtime issues, which is about a 7% improvement. We did not cause the disappearance of any true positive warning. The heuristic was retained in the RTEHunter system.

# 4 The Null Constraint Solver

## 4.1 Overview

The example we have discussed in Section 2.2 gives a quick insight into how the symbolic execution is preformed in the classical way. Both the presented source code and its conditional statements are simple, therefore, maintaning the path condition is easy. However, for more complex programs, the symbolic execution tree will be much larger. The number of branches grows almost exponentially with the number of symbolic conditional statements. This is what we call the path explosion problem. Due to the path explosion it is unlikely that symbolic engines can explore the whole symbolic execution tree exhaustively within a reasonable amount of time. Path explosion is connected to the difficulty that we are addressing in this research, namely the constraint solving problem. As previously mentioned, forming a path condition from the symbolic variables and checking its satisfiability at branching points are useful for pruning the symbolic execeution tree, filtering out unreachable program states and hereby false-positive warnings.

In practice, constraint solvers suffer from many limitations that affect computational time significantly. They consume a big portion of the overall runtime causing symbolic engines to scale poorly on bigger systems. Constraint solving optimizations have to be made to maintain a trade-off between accuracy and scalability.

Our team had gained experience in the use of constraint solvers with the predecessor of RTEHunter, as it was described in 2015 by Kádár et al. [58]. The open-source Gecode constraint satisfaction toolset was integrated to build and satisfy the constraint systems [42]. However, in RTEHunter we dismissed the introduced constraint solving mechanism due to resource usage issues. We decided to try a more lightweight, heuristical approach instead of a conventional constraint solver. The result of this idea was the null constraint solver. The results of this experiment were presented in a 2017 article [119].

The remainder of this chapter is organized as follows. The related work is discussed in Section 4.2. Section 4.3 describes our contribution, while Section 3.4 analyzes the results with an example.

## 4.2 Related Work

Path explosion and constraint solving are two main challenges in terms of the scalability of symbolic execution. Many optimization attempts were made. One of them is the previously mentioned concolic execution, which is the mixture of concrete and symbolic execution. For example, the KLEE[22], CUTE[87], jCUTE[69], and DART[45] symbolic engines use this method.

Another option is using the results of similar constraint expressions incrementally. Ramos and Engler [81] introduced the under-constrainted symbolic execution whose basic idea – executing functions directly – is similar to the method by method approach implemented in RTEHunter.

The Multiplex Symbolic Execution (MuSE) [109] system also battles to improve the scalability of symbolic execution. MuSE generates test inputs from the partial solutions during constraint solving, and uses them to explore multiple paths by solving the constraints only once. MuSE was implemented for the KLEE and JPF symbolic execution engines and tested with three constraint solving methods. The results showed a one or even two order speedup while reaching the same coverage.

Avgerinos et al. intruduced a method called 'veritesting' [14] to mitigate the difficulties of solving constraint formulas. During veritesting, the execution alternates between dynamic and static symbolic execution (SSE) [1]. The dynamic symbolic execution not only helps in solving the formulas produced by SSE, but it allows the handling of system calls and indirect jumps. Veritesting was implemented in a tool called MergePoint, which is used to automatically check all programs of a Linux distribution. They found that veritesting increases the number of bugs found, node coverage, and path coverage. Java Ranger [88] utilizes this method for Java programs. It is an extension for the Symbolic Pathfinder (SPF). The tests showed that the running time and number of execution paths were reduced by a total of 38% and 71%, respectively.

## 4.3 Contribution

The basic idea of this simplified solver is that we store constraints referring only to the null value of a symbolic variable. Constraints only express whether a symbolic object is null or not, therefore, only null assignments and null value checks are considered during the build-up of the path condition. It is clear that the expressions will remain quite simple and checking the feasibility is also very easy.

Listing 4.1 illustrates the operation of the solver. Let us consider that the symbolic execution is started with the `getHashCode()` method in line 8. This function calculates a hash code regarding the value of a custom typed class member, `data`. The symbolic execution tree built up during the execution can be found in Figure 4.1. Figure 4.1 does not show the details of the `toString()` method of the `CustomType`, and the execution of the `hashCode()` method is only indicated with a curved line. The path condition of each path is presented in transparent, black-bordered boxes next to the corresponding program states. For simplicity, it only contains constraints for the symbolic member variable, `data`, and is only presented when there is a change, namely after conditional statements involving variable `data`. It can be seen that on the `false` branch of a symbolic `if` statement the negation of the logical expression is added to the

[1]SSE is a verification technique for representing a program as a logical formula.

PC. When investigating the source code and the execution tree it is obvious that if the `getDataStr()` is called from the `getHashCode()` function, the conditional statement in line 4 can never be true. Consequently, the program states colored with yellow are unreachable. This unreachability is expressed in the PC too: it is infeasible.

```
 1  class Test{
 2    private CustomType data;
 3    public String getDataStr() {
 4      if(data == null)
 5        return null;
 6      return data.toString();
 7    }
 8    public int getHashCode() {
 9      int code = 0;
10      if(data != null)
11        code = getDataStr().hashCode();
12      return code;
13    }
14  }
```

**Listing 4.1:** Sample code

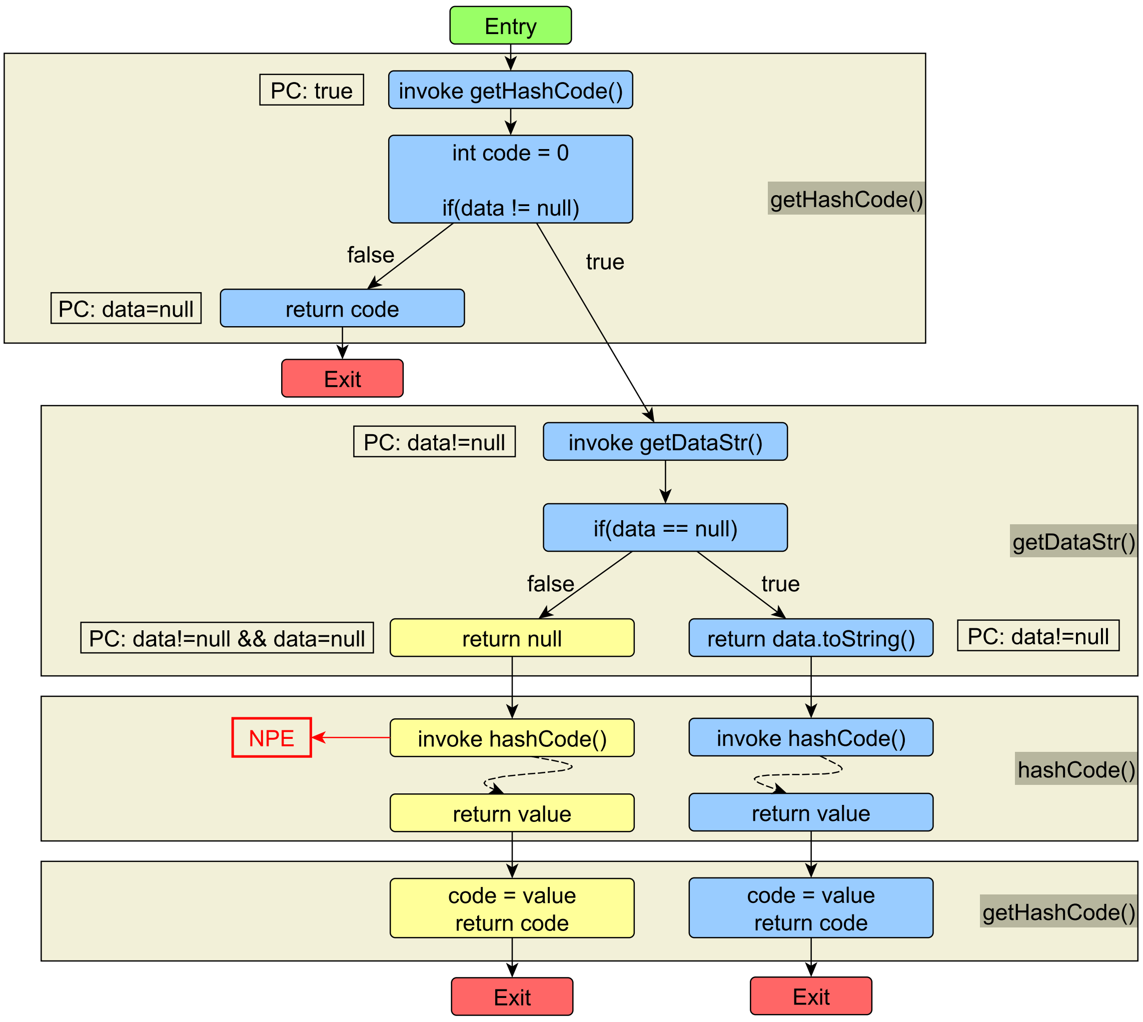

**Figure 4.1:** The simplified symbolic execution tree of Listings 4.1

The null constraint solver will notice on the `true` branch of the `if` statement in Line 11 that the member variable `data` is anything but null, so when the second conditional statement is reached in Line 4 the infeasibility can be detected with almost zero effort. That is, this solver tracks whether or not the value of a Java reference is null throughout its lifetime. This approach did not notably alter the computational time requirements of RTEHunter, whose previous version entirely lacked a constraint solving mechanism. We lost the arithmetic constraints, however, this seemed to be a viable trade-off.

## 4.4 Results

The null constraint solver was tested on almost the same Java systems that were used for the evaluation of the static init block handling as well. Only the largest project was replaced by a smaller one so that the analysis would not take that much time. For this reason, there is a difference in the number of errors between Table 3.2 and Table 4.1, even though the two heuristics were built on top of each other.

| **Warning type** | **Original RTEHunter** | **Improved RTEHunter** |
|---|---|---|
| NullPointerE. | 3,102 | 3,093 |
| NegativeArraySizeE. | 24 | 23 |
| DivideByZeroE. | 167 | 167 |
| ArrayIndexOutOfBoundsE. | 681 | 675 |

**Table 4.1:** The results of the original and improved RTEHunter

To see the improvement, we executed both the original, constraint solverless and the improved versions of RTEHunter on the selected Java systems. The results are presented in Table 4.1. As it can be seen there is a decrease in the number of *NullPointerExceptions* (NPE), *NegativeArraySizeExceptions*, and *ArrayIndexOutOfBoundsExceptions*. The differences in the numbers obscure that in fact 16 NPEs were eliminiated and 7 new were introduced. RTEHunter stops the execution along a path when a runtime failure is detected, therefore, it is obvious that the new warnings showed up because by eliminating unreachable paths the symbolic engine could explore other paths in the symolic executional tree before reaching the executional limits. This explanation was verified by manual examination.

We also manually examined why the *NegativeArraySizeException* and *ArrayIndexOutOfBoundsException* error messages disappeared. These were also false-positive warnings which were related to the non-storage of null values. Section 4.4.1 shows an example for them.

### 4.4.1 Example

This subsection presents a concrete example of the removed warnings.

Listing 4.2 shows an example for the eliminated *NegativeArraySizeException*. The example is taken from the Apache Tobago[2] system. A warning is reported for Line 15, although it is clear that if parameter `node` holds a null value, the execution will continue on Line 5. If parameter `node` is not null, variable `n` will be a non-zero value, therefore,

[2]`https://github.com/apache/myfaces-tobago`

no *NegativeArraySizeException* will be thrown on Line 15. Our null constraint solver was able to track the possible values of the `node` and prune the infeasible path.

```
 1 public class TreePath implements Serializable{
 2
 3   public TreePath(TreeNode node) {
 4     if (node == null) {
 5       throw new IllegalArgumentException();
 6     }
 7
 8     final List<TreeNode> list = new ArrayList<TreeNode>();
 9     int n = 0;
10     while (node != null) {
11       list.add(node);
12       node = node.getParent();
13       n++;
14     }
15     path = new int[n - 1];
16
17 }
```

**Listing 4.2:** The false-positive NegativeArraySizeException from Apache Tobago

Another example is presented in Listing 4.3. This simplified code is part of the Apache Ofbiz system[3]. Here, a *NullPointerException* is reported for Line X because of the `orderByElements.size()` call. However, that is impossible if we consider the surrounding code. The `orderByElements` variable will be null if the `UtilXml.childElementList(element, "order-by")` call returns a null value. This is possible if and only if the `element` method parameter holds a null value. Nevertheless, by tracking the super call of the `OrderMapList`, it appears that the `element` parameter has already been used in the constructor of `MiniLangElement` , therefore, its value cannot be null. The null constraint solver was able to filter this path without building and solving high complexity path conditions.

```
 1 public final class OrderMapList extends MethodOperation {
 2
 3   public OrderMapList(Element element, SimpleMethod ↩
         simpleMethod) throws MiniLangException {
 4     super(element, simpleMethod);
 5     ...
 6     List<? extends Element> orderByElements = UtilXml.↩
           childElementList(element, "order-by");
 7     if (orderByElements.size() > 0) {...}
 8   }
 9 }
10
11 public class UtilXml {
12   public static List<? extends Element> childElementList(↩
         Element element, String childElementName) {
13     if (element == null) return null;
14       ...
15       return elements; //not null
16     }
17 }
18
```

[3]https://github.com/apache/ofbiz

```
public abstract class MethodOperation extends MiniLangElement ↩
   {
  protected MethodOperation(Element element, SimpleMethod ↩
      simpleMethod) {
    super(element, simpleMethod);
  }
}

public class MiniLangElement {
  public MiniLangElement(Element element, SimpleMethod ↩
      simpleMethod) {
    this.lineNumber = element.getUserData("startLine");
    this.simpleMethod = simpleMethod;
    this.tagName = element.getTagName().intern();
    }
}
```

**Listing 4.3:** The false-positive NullPointerException from Apache Ofbiz

## 4.5 Conclusion

In this research, we also intended to improve RTEHunter with heuristic methods. This time, our goal was to fill the missing constraint solving mechanism with a low-resource solution. Constraint solving is an essential part of symbolic execution, it can be used to decide whether an executional path is feasible. Infeasible paths are pruned from the symbolic executional tree. However, like everything, it has its downsides. It can be very resource intensive and it may slows down the symbolic execution engine so much that it can traverse only a small slice of possible execution paths in a meaningful amount of time. There are a plenty of heuristic solutions that try to overcome this bottleneck. Since RTEHunter's constraint solving mechanism was removed due to its ineffectiveness, we decided to experiment heuristic solution as well. We have implemented the null constraint solver, which only takes into account the null or non-null state of variables. We tested the solution on more than two hundred Java systems and manually checked the results. 16 NPEs were eliminiated and 7 new were introduced, whereas the execution could have continued on some executional branches that were halted due to a false-positive warning. No change in runtime was observed. Our conclusion is that not only did we eliminate some serious false-positive warnings, but also discovered several true positive runtime issues. The heuristic solution was later further developed in RTEHunter by others.

# 5
# Summary of Part I

The first part of my research work was about symbolic execution, more specifically about improving the functionality of an existing Java symbolic engine called RTE-Hunter. Symbolic execution could be a quite powerful quality assurance technique as it theoretically executes every possible execution path of a program. It can be considered a kind of generalized testing. During regular execution, the variables of the program have concrete values, which means the program follows a specific execution path determined by these values. The drawback of symbolic execution is that it requires excessive resource usage compared to normal execution. There are several strategies to reduce these demands. For example, constraint solving can be used to filter out unsatisfiable execution paths. Another disadvantage of symbolic execution is that certain elements, such as the behaviour of the operating system, are not trivial to model symbolically. For this reason, the result may not be as accurate as expected.

The goal of this thesis point was to find solutions for RTEHunter that would increase the accuracy of its symbolic execution without significantly increasing its resource requirements. One of the heuristics was related to the handling of static initialization blocks in Java. For RTEHunter, which performs execution per method, handling these symbolically is not quite simple, as it is not exact when and in what order the blocks should be executed. Because of this, static initialization blocks were completely neglected during execution leading to plenty of false-positive error messages, especially `NullPointerExceptions` (NPE). My heuristical solution was a filtering mechanism that fits well with RTEHunter's method-by-method-based implementation. When an NPE arose, we simply checked if the variable associated with this warning was initialized in the static initializer blocks of its class. If yes, the warning was treated as a false-positive. The approach was tested on 209 various-sized Java systems. Of the original 3,369 NPE warnings, 242 disappeared and were verified as false-positives by manual inspection. In addition, 7 true positive warnings appeared because RTEHunter was able to continue the execution on branches that were previously terminated due to the false-positive NPEs.

The other heuristic was a low-resource constraint solving mechanism called the null constraint solver. It tracks only if a reference is null or not and does not account for arithmetic variables. RTEHunter's previous version entirely lacked a constraint solving mechanism. To see the improvement, we executed both the original and the improved versions of RTEHunter on 209 Java systems (the largest test system was replaced for this evaluation). After manual validation, we concluded that we not only eliminated 16 serious false-positive warnings (out of 3,102), but also discovered 7 true positive runtime issues.

In this thesis point, my goal was to develop methods to increase the accuracy of the warnings of RTEHunter without further increasing the resource requirements of the already computationally intensive process. Analyses performed on more than two hundred systems have confirmed that this aim was successfully achieved.

# Part II

# Comparison of Static Call Graph Builder Tools

# 6 The Significance of Call Graphs

## 6.1 Motivation

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 $b$. They are the main building blocks of modeling interprocedural control and data flow; their soundness can greatly affect the results of subsequent analyses. Static code analyzer tools that help programmers produce more maintainable code utilize such call graphs. They come in handy for program visualization and program understanding. Moreover, they can also be employed for code optimizations and refactoring.

Because call graphs play such an important role in static analysis, for example in symbolic execution, we decided to focus our research interest on Java call graph generation. Java is an object-oriented, general purpose programming language that is still widely used thanks to its platform independent nature. One important area of application is client-server based web applications. Platform independent behavior is ensured by the fact that the Java source code is only compiled into byte code which runs on a Java Virtual Machine (JVM) regardless of the underlying architecture. Java supports inheritance, polymorphism, and has dynamic capabilities, such as reflection.

Our research has highlighted how challenging the richness of the Java language can be in generating call graphs. There are numerous aspects to consider depending on the future purpose of the generated call graph. Due to the size of the task, we developed our research through several articles and works. Initially, we collected five Java static analyzers to generate call graphs. We examined their abilities, the differences between their graphs etc. The results were published in a short, comparative article [116]. However, we soon realized that the topic requires more in-depth study. We have expanded this work by including a new static analyzer and eliminating the tools that could not provide sufficient results on large projects [113]. The comparison of the analyzers' outputs required special attention. It was not as simple as it first seemed; we had to develop several heuristic approaches to refine the comparison. This work was described in an article [115].

This chapter provides related work in Section 6.2. In Section 6.3 an example is presented to illustrate the challenges of call graph creation. Section 6.4 enumerates the six static analyzers that we selected for the comprehensive study. Chapter 7 details the difficulties of making the tools' outputs comparable, while Chapter 8 summarizes the findings of the in-depth comparison.

## 6.2 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 [102], [35], [28], [100]. 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. [33], who created and mined weighted call graphs to achieve more precise bug localization. Liu et al. [66] constructed behavior graphs from dynamic call graphs to find non-crashing bugs and suspicious code parts with a classification technique.

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 [15], [82]. A call graph is accurate if it contains those exact 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)* [32], *Rapid Type Analysis (RTA)* [15], *Hybrid Type Analysis (XTA)*[96], *Variable Type Analysis (VTA)* [93].

Another important question during call graph creation is the handling of library calls [10]. 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. [82] 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.

There are many comparative studies available about call graph creation. Grove et al. [49] implemented a framework for comparing call graph creation algorithms and assessed the results with regard to precision and performance. Murphy et al. [75] 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 [54]. Antal et al. [13] 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. [96] 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 [64] compared static call graphs generated by Soot [85] and dynamic call graphs created with the help of the *J [8] 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. [83] dealt with the unsoundness of Java call graphs. They compared the call graph creator capabilities of two analyzer tools, WALA and Soot. They evaluated the 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. The article highlights how well the given tools handle the different language elements of Java. An assessment suite for the comparison of different call graph tools is proposed as well. Our comparative work is similar to their study, however, we provide a full scale of factors that can cause ambiguities in the call graph creation through the in-depth analysis of six call graph tools.

## 6.3 Call Graph Example

This section presents some difficulties of call graph creation through an example. Listing 6.1 shows a Java snippet from which we want to create a call graph. It contains a class named `Sample` with a `main` method. It has a static method on Line 11 and a method on Line 18 that has a polymorphic parameter. The `Base` class (which is not present in the sample code) has several derived classes such as, `Derived1` and `Derived2`, therefore, the actual parameter of the `parameterAsBase` method can either be a `Base`, `Derived1`, or `Derived2` object. We can see an example of a polymorphic method call on Line 6 and 7. There is also an `unused` method in the `Sample` class which is never called in the entire program.

Figure 6.1 shows a possible call graph generated for the sample code. The black nodes represent Java library methods, such as `System.out.println()` and `java.util.Random.nextInt()`, and the gray nodes correspond to the constructor (e.g. initializer) methods. We can see every node and edge that is logical and evident for us. We can see the constructor calls from the `main` method, the recursive call of the `staticTask`, and even the separate subgraph of the `unused` method. However, this graph is not so evident for a static analyzer. Take a closer look at the outgoing edges of the `parameterAsBase` method. There are three of them. We wanted to find out why. The static type of the parameter is `Base`, therefore an analyzer could choose to display only the call to the `Base.polymorph()` method. Another solution is that the analyzer tries to approximate the dynamic type of the parameter and connect to their `polymorph()` as well. The brute force approach represents the `polymorph()` method of every subclass of the Base class. Another uncertainty is the unused subgraph. Some static analyzers can choose to eliminate it, as it is not connected to the main method. Another potential inaccuracy: where are the initializer nodes of the Object class? In Java, every class is derived from the Object class, therefore, its constructor is also called implicitly. Would it be useful to include them in the generated graphs or it would be unnecessary noise? And what about the static initializer blocks? The answer to these questions depends on the purpose for which the call graph was generated.

For comparison, Figure 6.2 shows another possible call graph consructed for the sample code. It represents the static initializers with red-edged nodes and the calls to the Base and Object constructors as well. Naturally, the static initializer nodes can call other methods, constructors as well, making the graph even larger. Some tools might connect them to the constructors of their class, but these edges are not presented in this figure.

```java
public class Sample {

  public static void main(String[] args) {
    Base b = new Derived1();
    Derived2 d2 = new Derived2();
    parameterAsBase(b);
    parameterAsBase(d2);
    staticTask(3);
  }

  public static void staticTask(int i) {
    if (i > 0)
      staticTask(i - 1);
    else
      System.out.println("Task finished");
  }

  public static void parameterAsBase(Base b) {
    b.polymorph();
  }

  public static void unused() {
    java.util.Random r = new java.util.Random();
    int i = r.nextInt();
  }
}
```

**Listing 6.1:** Input for call graph generation

## 6.4 Selected Static Analyzers

We studied numerous static analyzer tools for Java to decide whether they could generate – or could be easily modified to generate – call graphs. We aimed for widely available, open-source programs from recent years, which could analyze complex, real-life Java systems. The diversity of the tools was another important aspect of our selection criteria. We involved tools that provide a direct interface for call graph creation, whilst, in other cases, the graph had to be extracted directly from the inner representation of the analyzer. The investigated analyzers can also be categorized by whether they work on source or byte code, which, of course, affects their results. Most

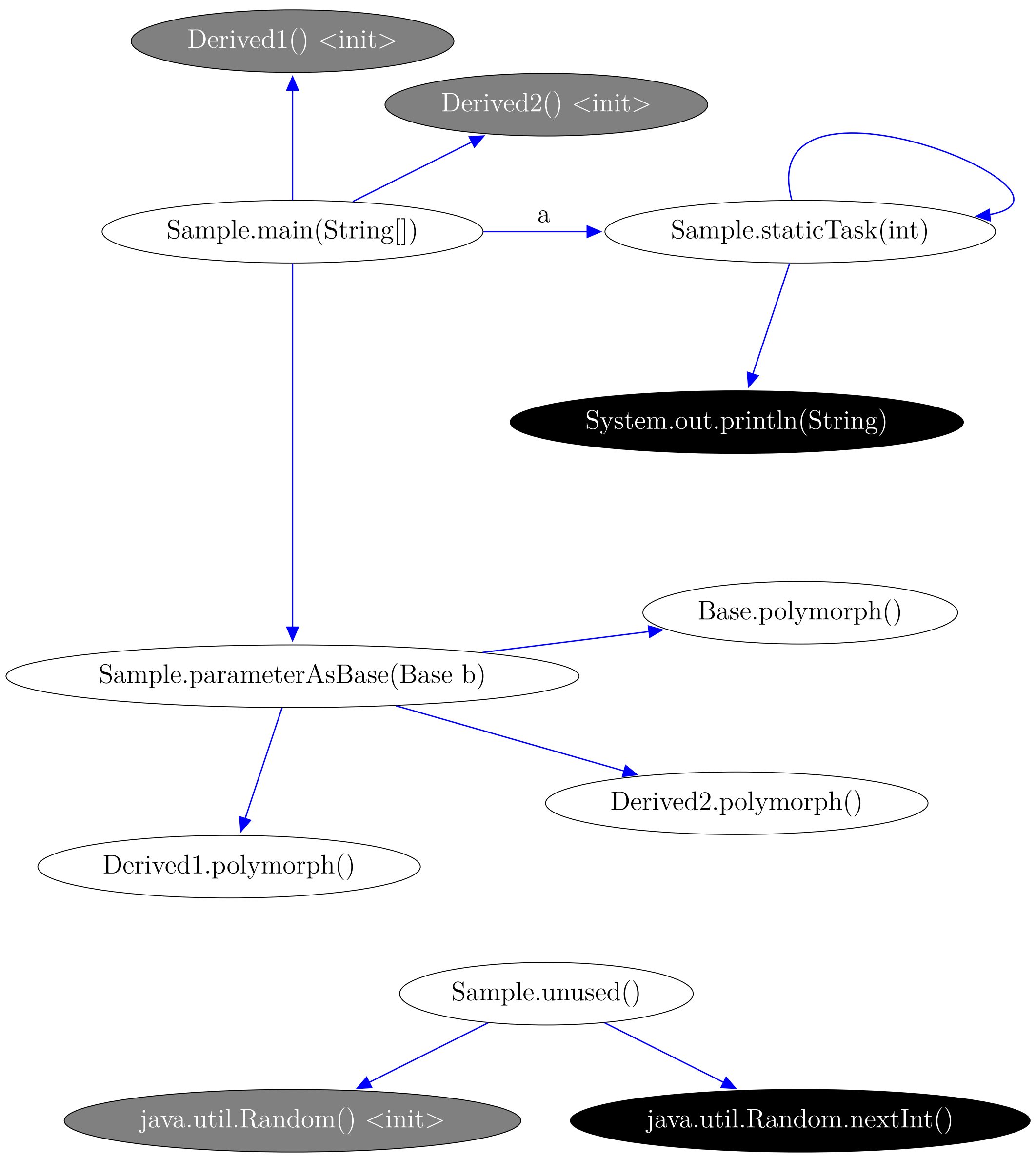

**Figure 6.1:** Call graph of the sample code

of the tools are command line based, although an Eclipse plug-in based solution was also examined. The selected analyzers support several call graph-creation algorithms, which greatly influences the characteristics, the accuracy, and the size of the generated graph. It is the application that determines what type of call graph is the most useful; sometimes a small and less accurate call graph is better, while, in other cases, a large and precise one is needed. The goal of this research is not to compare the output of these call graph-builder algorithms, but to pair the corresponding graph sections, which is the basis for further comparative studies. Therefore, we considered the algorithm only as an attribute of the given tool. We eliminated the tools that did not give enough information to reconstruct the caller-callee relationships between compilation units without major development (e.g., JavaParser [31]). The selected tools, the analyzed sources and the results are available as an online appendix[1].

The description of the six tools that were selected for the comparison is presented

[1]`http://www.inf.u-szeged.hu/~pengoe/research/StaticJavaCallGraphs/`

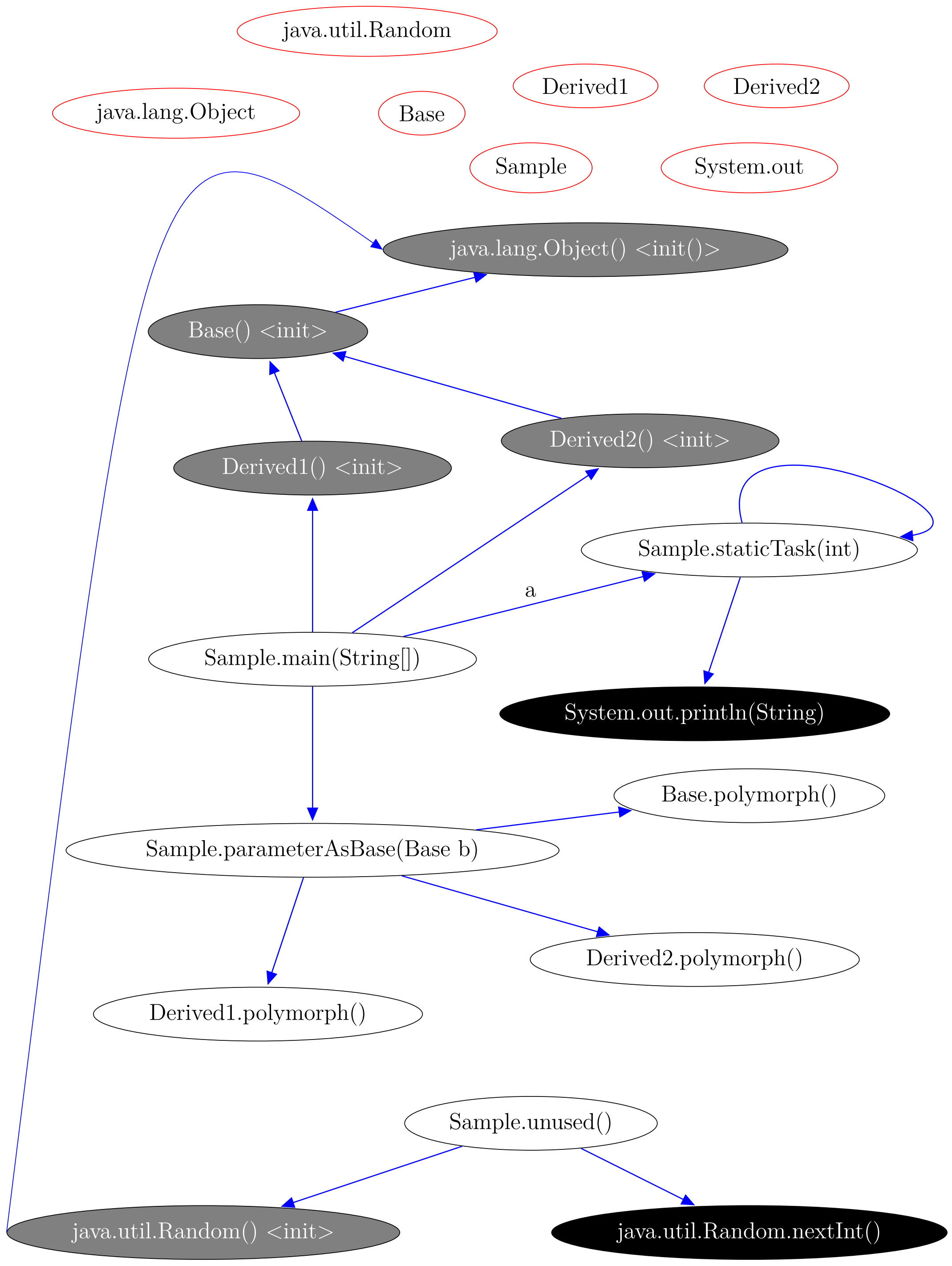

**Figure 6.2:** Call graph of the sample code

below.

### 6.4.1 Java Call Graph

The Java Call Graph (JCG) [43] is an Apache BCEL [1] based utility for constructing static and dynamic call graphs. It can be considered a small project, as it only has one major contributor, Georgios Gousios, whose last commit is from April, 2017. 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*[2] code. As a result, the call graph does not include calls from code segments that are never executed.

### 6.4.2 SPOON

SPOON [78] 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[3] of the source code and builds an AST-like metamodel, which is the basis for these further analyses and transformations. We extracted the call graph of our project by traversing this internal representation and collecting every available invocation information. The library is well-documented and provides a visual representation of its metamodel, which helped us in thoroughly studying its structure.

### 6.4.3 WALA

WALA [6] 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. WALA has a built-in call graph generation feature with a wide range of graph building algorithms. We used the *ZeroOneContainerCFA* graph builder for our experiments, as it performs the most complex analysis. It provides an approximation of the Andersen-style pointer analysis [12] with unlimited object-sensitivity for collection objects. The generator has to be parameterized 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 the 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.

### 6.4.4 OpenStaticAnalyzer

OpenStaticAnalyzer (OSA) [5] is an actively maintained, multi-language static analyzer framework developed by the Department of Software Engineering at the University of Szeged. It calculates source code metrics, detects code clones, performs reachability analysis, and finds coding rule violations up to Java 8. Other languages such as Python and C# are supported as well. It performs a directory analysis of the source code. This can make the analysis more precise, as generated files will be handled too. Similarly to the above mentioned SPOON implementation, we extracted the call graphs by processing the AST-like inner representation of OSA.

### 6.4.5 Soot

Soot [85] 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

[2]Unreachable code will never be executed as there is no control flow path to it from the entry point of the program.

[3]The static analyzer processes every Java file in a given root folder recursively.

9 and works on the compiled binaries. Although its latest official release was in 2012, the project is active with regular commits and nightly builds. Like WALA, it also has a built-in call graph creator functionality. We used the SPARK framework, which employs a points-to analysis algorithm during construction.

### 6.4.6 Eclipse JDT

The Eclipse Java development tools (JDT) [3] is one of the main components of the Eclipse SDK [2]. It provides a built-in Java compiler and a full model for Java sources. We created a JDT based plugin for Eclipse Oxygen that even supports Java 10 code, to extract the call graph from the extensive, AST-like inner representation.

# 7 A Preparation Guide for Java Call Graph Comparison: Finding a Match for Your Methods

## 7.1 Motivation

The way to compare the capabilities of call graph builder tools is through comparing their generated call graphs. Due to the increasing number of extremely large graphs and their wide usability, there are many algorithms and metrics available for comparing general directed and undirected graphs [9, 63, 68, 99]. However, these methods cannot be directly applied to call graphs, especially if they were produced by different analyzer tools. Call graphs are directed graphs whose nodes correspond to the methods in the source code. Even if the structure of two call graphs is isomorphic, they can be considered completely different because of the labeling of the nodes. Therefore, to make them comparable, we first have to find a mapping, which is the aim of this research.

There are several proposals for comparing labeled graphs if a mapping is already present. Champin and Solnon defined a similarity with respect to a given mapping between two graphs that have multiple labels on both their nodes and edges [26]. A graph is described by the set of all of its features, e.g. the set of node-label and edge-label pairs. The similarity measure is calculated based on a simplified version of Tversky's formula [98]. They also proposed an algorithm for finding the best mapping for reaching the maximum similarity, which provides a qualitative description of the differences between the two graphs.

There are algorithms available for the exact purpose of comparing labeled graphs that share the same node set [104]. In the case of call graphs that were produced by different tools and algorithms this condition cannot be ensured. The most simple solution to compare graphs with the same node set is to handle the adjacency matrices as vectors and calculate an edit distance, e.g. the number of different edges [34, 86]. Wicker et al. introduced a dissimilarity measure for graphs like these based on their eigenvalues and eigenvectors [104], which takes into account the global graph structures as well.

As it was already mentioned in the previous chapter, the precision and structure of the call graphs greatly depend on the algorithms that the builder tools used. If several call targets are possible for a given call site, more examination is needed to determine which edges should be connected. There are context-dependent and context-independent solutions; naturally, the choice influences the result. 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 starts only from the `main` method or a few entry points instead of starting from every method of the analyzed source code. This might lower the accuracy as well. Context-independent methods for object oriented languages can be improved with the following algorithms: *CHA* [32], *RTA* [15], *XTA*[96], and *VTA* [93]. In case of the comparison of these call graph building strategies [19, 48, 49], node matching is usually not an issue because the algorithms are implemented in the same environment, and language elements are handled similarly. The nodes of the produced call graphs are the subset of each other's node set with the same naming convention, therefore, the main difference comes from the number of edges.

In this research, we considered the results of static analyzer tools, meaning that we worked with the so called static call graphs. However, call graphs can be composed with dynamic tools as well from the actual executions of the analyzed program. Lhoták [64] compared static call graphs generated by Soot [85] and dynamic call graphs created with the help of the *J [8] 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. The paper does not describe the difficulties of matching the nodes of the call graphs that were provided by different sources. The reason could be that Soot is a bytecode analyzer, therefore, its output is close to the output of a dynamic analyzer.

Murphy et al. [75] carried out a study about the comparison of five static call graph creators for C in 1996. The outputs of the analyzers were compared to a baseline call graph created by the GCT test coverage tool, which was based on the GNU C compiler. They identified significant differences in how the tools handled typical C constructs, like macros. The mapping of the graphs was complicated by the handling of which files were involved in the analysis. They applied a filtering mechanism to solve this. Other difficulties of the matching mechanism were not discussed, although C functions are clearly identified by their name only.

Based on the literature review, we came to the conclusion that no one has yet described the kind of graph comparison we needed. To compare the call graphs that were generated by different tools for the same source code, we have to match the nodes – i.e. the methods – that correspond to each other, and then evaluate what types of methods and calls were found by each tool. However, matching the methods to each other is challenging, since it is not certain that all tools will find the same methods, or they might name them differently. The next section illustrates these obstacles of the method pairing mechanism and our step-by-step improvements with results. The research is concluded in Section 7.3.

## 7.2 Refining the Pairing Mechanism

In order to compare the call graphs generated by different tools, we need to identify the corresponding nodes – the targets of the potential invocations – in multiple graphs. However, there are a lot of elements that could cause differences in call graphs, as tools process language elements differently. In this section, we discuss what attempts we made to handle these differences, and what were the benefits and downsides to each approach. The process is illustrated through the Apache Commons Math 3.6.1[1] project (208,876 KLOC). We are only using one project as an example, since our aim is to showcase the process itself, not to compare data. More analyzed projects are presented in the online appendix[2].

### 7.2.1 Basic Name Pairing

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. The return value is not required for the identification, however, we encountered one case where this did not prove sufficient. According to the Java standard, overridden methods can differ in return type if the return-type-substitutability is satisfied, such as, the child class specializes the return type to a subtype. Some tools represent both the specialized and the not-specialized methods for a child class, although they only connect edges to one of them. Therefore, these rare cases could be easily detected.

Naturally, each static analyzer tool produced a slightly different output. For example, OSA and WALA use the standardized naming convention of Java (JNI notation)[3], while others employ their own notation system. In the case of the analyzer for which call graphs were extracted manually from the internal representation, we also tried to follow the JNI notation, although it was not always desirable. It was easier to use an output close to the tools' representation and then handle the discrepancies with the pairing program than to implement a fully standard JNI notation. To illustrate the heterogenity, Figure 7.1 shows the different textual representations of the `boolean method(Child c, Class <?>... objects)` method in the `ForParser` class. This method takes a `Child` object as a first parameter and a variable number of `Class<?>` objects.

These textual differences in the representation were easy to deal with. Another notable difference is the representation of the constructor methods. For example, some tools denote constructor methods with the class name, whilst others tag them with the name `<init>`.

A loading module had to be created for the output of each tool. We developed a common representation for the Java methods, and as a first step of the comparison we transformed every call graph into this unified representation. This is the basis for our first node pairing approach, the Basic Name Pairing.

In this algorithm, we only used the method names produced by the static analyzers as basis for the method pairing. Only the constructor methods and other not significant representational differences were subject to the name unification process. Table 7.1

[1]`http://commons.apache.org/proper/commons-math/`

[2]`http://www.inf.u-szeged.hu/~pengoe/research/StaticJavaCallGraphs/`

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

- OSA: `ForParser.validMethod(LChild2;[Ljava/lang/Class;)Z;`
- Soot: `<ForParser: boolean method(Child,java.lang.Class[])>`
- SPOON: `root.ForParser.method(Child,java.lang.Class[])`
- JCG: `M:ForParser:method(Child,java.lang.Class[])`
- WALA: `ForParser.method(LChild;[Ljava/lang/Class;)Z;`
- JDT: `ForParser.method(Child;java.lang.Class<?>[]):boolean`

**Figure 7.1:** Various textual representations of a method with variable number of parameters

summarizes the results of this initial attempt on the Commons Math project. The diagonal elements in bold show the number of different methods found by each static analyzer tool. Every other cell in a row is a percentage that displays how many percent of the given tool's methods was found by the tool in the column.

| | Soot | OSA | SPOON | JCG | WALA | JDT |
|---|---|---|---|---|---|---|
| Soot | **813** | 78.84% | 81.06% | 87.08% | 80.93% | 70.73% |
| OSA | 7.59% | **8,447** | 99.96% | 96.38% | 82.38% | 73.56% |
| SPOON | 7.63% | 97.81% | **8,633** | 95.75% | 82.09% | 72.01% |
| JCG | 7.11% | 81.81% | 83.07% | **9,951** | 77.21% | 61.67% |
| WALA | 7.70% | 81.43% | 82.93% | 89.90% | **8,546** | 68.50% |
| JDT | 6.11% | 66.04% | 66.07% | 65.22% | 62.21% | **9,410** |

**Table 7.1:** Results of the basic name pairing

It can be seen that the column of Soot contains quite low values. Soot found very few methods compared to the other analyzers, therefore, its highest possible percentage is at about 10%. The reason for this discrepancy lies in the algorithmic differences between the tools, however, analysing this not the subject of the current research.

### 7.2.2 Anonymous Transformation

An anonymous class is an inner class without a name. It is useful when the programmer needs one instance of a class or interface with only certain overridden methods, so the actual subclass creation can be avoided. Lambda methods can be considered anonymous, however, most analyzers denote them with their interface name. Anonymous source code elements have a non-standardized, compiler generated name, meaning that static analyzers can name the same code element differently. Inner classes have a '$' sign in their name appended right after the name of the outer class. The '$' sign is followed by the name of the inner class. In case of anonymous classes, a number is present after the '$' sign, however, the numbering is not consistent among the compilers and analyzer tools. Both global, project-wise numbering, and class level numbering is possible. The order of the numbering can also make a difference in the output of the tools. It is clear that our basic pairing approach that was introduced in the previous subsection is not sufficient for pairing anonymous code elements.

The challenge is illustrated with the example that can be seen in Listing 7.1. In the `main` method of the `ForParser` class an anonymous class is created by overriding the `Outer` interface. Inside the `m1` method's inmplementation another anonymous class is defined based on the `Inner` class.

```java
public class ForParser {

 public static void main(String[] args){
   Outer a1 = new Outer () {

        public void m1() {

          Inner a2 = new Inner(){

             public void m2() {

               System.out.println("I'm anonymous");

             }
          };
        }
     };
  }
}
```

**Listing 7.1:** Sample code containing a nested anonymous class

The name of the `m2` method according to each tool is presented in Figure 7.2.

- OSA: `ForParser$1.m2()V`
- Soot: `<ForParser$1$1:  void m2()>`
- SPOON: `root.ForParser$1.m2()`
- JCG: `M:ForParser$3:m2()`
- WALA: `ForParser$3.m2()V`
- JDT: `ForParser$3$3.m2():void`

**Figure 7.2:** Various representations of a method of nested anonymous class

It can be seen that a heuristical approach is needed to match nodes like these in the call graphs. The so called *anonymous transformation* was introduced as a solution for handling anonymous code elements. Table 7.2, which is constructed similarly to Table 7.1, shows the results of the method pairing improved with anonymous transformation. The green cells highlight the higher percentages compared to Table 7.1. The transformation simply means that during the name unification process we replace the varying number after the '$' sign with a constant string, and then the name-based pairing is performed. This means that multiple anonymous elements in a class will be aggregated into one, which is the explanation for the smaller method numbers in the diagonal cells of Table 7.2. For example, if a class has multiple anonymous classes, all of them will be transformed for the unified anonymous class, causing a loss in the accuracy of the pairing. For projects that do not rely on anonymous classes very much – i.e. in a class there is at most one anonymous element – this heuristical approach is acceptable.

| | Soot | OSA | SPOON | JCG | WALA | JDT |
|---|---|---|---|---|---|---|
| Soot | **808** | 79.21% | 81.44% | 87.00% | 80.82% | 73.02% |
| OSA | 7.68% | **8,330** | 99.96% | 96.95% | 82.76% | 76.30% |
| SPOON | 7.76% | 98.15% | **8,484** | 96.53% | 82.64% | 75.05% |
| JCG | 7.19% | 82.61% | 83.78% | **9,776** | 76.80% | 64.00% |
| WALA | 7.81% | 82.47% | 83.87% | 89.82% | **8,359** | 71.60% |
| JDT | 6.36% | 68.57% | 68.68% | 67.50% | 64.56% | **9,270** |

**Table 7.2:** Results of the anonymous transformation

### 7.2.3 Employing Line Information

It was a self-evident idea to include the line information in order to improve the accuracy of the method matching. In addition to the methodnames, we also used their position in the source code for pairing. However, we have soon found out that the line information does not provide a perfect solution for the problems of method pairing because it is not as consistent among static analyzers as it could be expected.

One obvious difficulty is that some of the tools process the source files themselves, while others work on the already compiled class files. Source code analyzers provide line information to the beginning and end of the method declaration. Byte code analyzers give the line information for the first statement of the method in question. In case of an empty method, the line information of the ending of the method declaration is present. This difference can be overcome by interval testing. Moreover, not every method has line information because they are compiler generated or they are part of the Java library. In other cases, only some of the tools can provide line information for a method. In addition to these difficulties, we realized that in very few cases tools provide the line information of the beginning of the class definition for some methods. These are inherited methods, whose definition is not part of the investigated source files. As a consequence, some methods that certainly differ have the same line information. We encountered this issue only during the analysis of the Joda-Time[4] project and could not reproduce it with manually created test cases.

Seeing these difficulties, it is clear that we cannot rely on line information blindly, because it would misguide the pairing mechanism. Therefore, the usage of line information was restricted only for anonymous and generic source code elements, whilst, for traditional methods, the name-wise pairing was used. The challenge of anonymous elements has already been discussed. In case of those, we used only the line information for matchmaking. Generic elements raised a new type of issue that is introduced in the next section.

Table 7.3 shows the improvement of results compared to the basic name-wise pairing that is summarized in Table 7.1. In case of Soot and WALA we can see a slight decrease in the number of methods. It is because these tools - erroneously - provided the same

[4]`https://github.com/JodaOrg/joda-time`

line information for some anonymous methods, therefore, they could not be handled separately. The approach best improved the pairing of the JDT as this is the most reliable tool for providing line information.

| | Soot | OSA | SPOON | JCG | WALA | JDT |
|---|---|---|---|---|---|---|
| Soot | **810** | 79.14% | 81.36% | 87.04% | 80.86% | 72.72% |
| OSA | 7.59% | **8,447** | 99.96% | 96.38% | 82.38% | 75.53% |
| SPOON | 7.63% | 97.81% | **8,633** | 95.75% | 82.09% | 74.12% |
| JCG | 7.08% | 81.81% | 83.07% | **9,951** | 76.58% | 63.50% |
| WALA | 7.72% | 82.03% | 83.54% | 89.83% | **8,483** | 71.34% |
| JDT | 6.26% | 67.80% | 68.00% | 67.15% | 64.31% | **9,410** |

**Table 7.3:** Results of the line information - based transformation (anonymous)

### 7.2.4 Strategy for Handling Generic Elements

Java generic classes and methods were introduced in JDK 5.0. They allow programmers to specify a set of methods and a set of types with only one method and class declaration, respectively. A single generic method can be called with arguments of various types. One important trait of generic classes is that they can be parameterized differently during instantiation. Generic type parameters can be bound, which restricts the types that are allowed to be passed.

Static analyzers represent generic elements in the call graph in various ways. Figure 7.3 shows the diversity of representations after the method name unification. It can be seen that the tools represent them with differing accuracy. Sometimes generic parameters are represented by the prototype that is present in the declaration, optionally involving the type restriction too (e.g., SOOT). In other cases, the type of the actual parameter is used, that is, the tool represented the same generic method with multiple nodes but with differing generic parameters.

The Java compiler applies type erasure to generic elements. This means that the compiler enforces type constraints at compile time. It replaces all type parameters in generic types and methods with their bounds or with `Object` if the type parameters are unbound. In Figure 7.3 parameter `T` is unbound, while parameter `K` is bound to `Child2` or its descendants. Most tools also rely on type erasure, however, WALA and Soot attempt to propagate the types of the actual parameters into the called method. In case of JDT, we collect information from the AST representation of the analyzed program, and we determine the type of the parameters 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.

The ideal solution would be to pair the corresponding generic methods to each other, but because of the variety of the notations, matching them only by the basic pairing process caused inaccuracies. Although the package, class, and method names are the same, even the number of parameters are the same, the type of the parameters can differ. Unlike in the case of anonymous methods, it is not always possible to decide whether a generic method is generic or not, based on its name alone. Therefore, the line information is needed to decide if two methods with the same name and number of parameters correspond to the same generic method. If the line information is the same as well, then the two nodes apply to the same generic method. This heuristical

- Declared method:
  `<T, K extends Child2> Generic2<Child2, Generic1<Child2> >methodGen(K c, Generic1<K> g, Class<?>...objects)`

- Usage:
  `methodGen(new Child2(), new Generic1<Child2>(), Integer.class)`

Representations:

- OSA
  `Generic2 methodGen(Child2,Generic1,java.lang.Class)`
- SPOON
  `methodGen(K extends Child2,Generic1,java.lang.Class[])`
- JCG
  `methodGen(Child2,Generic1,java.lang.Class[])`
- Soot
  `Generic2 methodGen(Child2,Generic1,java.lang.Class[])`
- WALA
  `Generic2 methodGen(Child2,Generic1,java.lang.Class)`
- JDT
  `Generic2<Interface,Generic1> methodGen(K,Generic1<Interface>, java.lang.Class<?>[])`

**Figure 7.3:** Various representations of a generic method

assumption has a threat to validity if the tool provides false line information. What is more, the pairing is not possible if no line information is given. However, combining line information with generic elements caused a new type of problem, which is summarized in Figure 7.4.

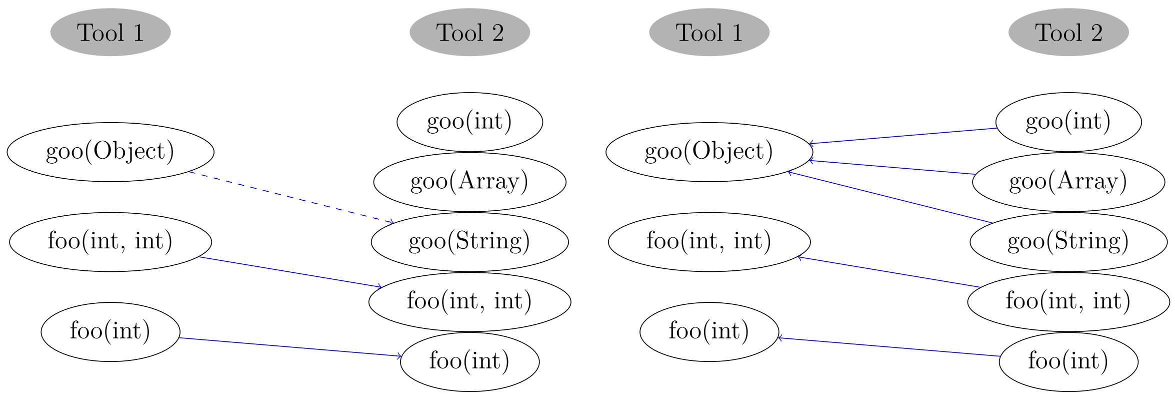

**Figure 7.4:** Pairing anomaly

Figure 7.4 shows two static analyzers, Tool 1 and Tool 2 (denoted by grey ellipses) and methods they detected during analysis (denoted by white ellipses). The analyzed source code contains a generic method, `<T> void goo(T t)` and two normal methods, `void foo(int a, int b)` and `void foo(int a)`. Tool 1 represents `goo` in the call graph with only one node. As there is no restriction on the type, the tool denotes the parameter type as an `Object`. In contrast to this, Tool 2 associates three nodes to method `goo` based on the type of the actual parameters it was called with. All `goo` nodes have the same line information. The matching of the `foo` methods is obvious, as both of them are represented with one node each. This is not the case with the pairing of method `goo`. The left side of the figure shows a possible way to match the nodes of Tool 1 to the nodes of Tool 2. The pairing of `goo` is denoted with a dashed line, as other matches would be possible if it was allowed to pair one method to multiple others. The right side of the figure shows the process int the opposite direction: the matching of the nodes of Tool 2 to the nodes of Tool 1. It can be seen that all `goo` nodes will be paired to the same node in the graph of Tool 1, because there is no other option. As a consequence, there is asymmetry in the results depending on the direction from which we start pairing the nodes.

The described pairing anomaly can be resolved in multiple ways. One solution is to use the results as they are, without any further modifications. This approach emphasizes the differences between the tools' capabilities. Another option is to only keep those node-matchings that can be found from both directions. Finally, we can collect every possible pairing from both directions and put them into a union. The union pairing was the solution we decided to use. Table 7.4 summarizes the results of this approach. The structure of the table is similar as before, the green cells highlight the higher percentages compared to Table 7.3. There is a decrease in the number of methods because we counted the corresponding generic methods as one.

| | Soot | OSA | SPOON | JCG | WALA | JDT |
|---|---|---|---|---|---|---|
| Soot | **805** | 79.50% | 81.74% | 87.45% | 81.24% | 73.42% |
| OSA | 7.58% | **8,442** | 99.96% | 96.38% | 82.41% | 76.21% |
| SPOON | 7.63% | 97.81% | **8,627** | 95.75% | 82.10% | 74.77% |
| JCG | 7.08% | 81.83% | 83.12% | **9,942** | 76.58% | 63.87% |
| WALA | 7.71% | 82.05% | 83.54% | 89.80% | **8,479** | 71.86% |
| JDT | 6.28% | 68.41% | 68.59% | 67.51% | 64.79% | **9,404** |

**Table 7.4:** Results of the transformation based on line information (anonymous and generic elements)

### 7.2.5 Remaining Differences

Table 7.4 shows that we could not achieve 100% pairing for the tools; a significant number of nodes remained unmatched. We manually investigated the root causes for this, in order to find possible ways to improve our pairing mechanism. However, our in-depth examination revealed that most of the unmatchings cannot be resolved. The reasons of the differences can be categorized as follows:

- A tool detects a method type that other tools do not represent
    - Soot represents much more static initializer nodes then the other tools.
    - WALA places more Java library nodes and calls into the generated call graphs.
    - SPOON represents Java static field initialization with a unique node.

- The methods that can be found in the bytecode slightly differ from the methods of the source code.
  - Bytecode analyzers (JCG, WALA, Soot) find compiler generated `access$XXX` methods, which cannot be paired with improper line information.
  - Source code analyzers only detect default constructors for `Enum` classes. Byte code analyzer tools represent the valid constructors with an `Integer` and a `String` parameter.
  - In the compiled sources, the methods of inner classes have an extra parameter, a reference to the outer class. This parameter is missing from the findings of the source code analyers.
- Algorithmic differences in the handling of polymorphic calls.
  - Tools that employ less accurate analysis techniques represent more interface and base class methods instead of the methods of the subclasses.
  - JCG represents inherited methods as the method of the child class, while other tools represent them as part of the base class.
- Exclusion of methods that do not have at least one method call. OSA and SPOON have this feature.
- Missing line information for anonymous and generic methods.

We concluded from our findings that our pairing mechanism could only be improved with more reliable line information.

## 7.3 Conclusion

This research was a necessary preliminary work for the quality comparison of the static call graph generator tools. Their capabilities can only be assessed if we compare what methods and calls are present in the generated call graphs. If the nodes of the call graphs are matched, then comparing the calls is a straightforward task. That is the reason why we paid so much attention to the unifying process of the methods.

We collected and, where necessary, modified six Java static analyser tools to generate call graphs for multiple large projects. By investigating the resulting graphs, we realized that the unification of method names is needed, in order to be able to match the corresponding nodes to each other. The unification process - and hence the pairing mechanism - has been refined in several steps. We highlighted two common language elements, the anonymous and generic methods, that needed careful consideration and made the improvement of the process necessary. Multiple solutions were proposed. One heuristical − but less accurate − approach for anonymous elements is the anonymous transformation. However, with line information they could be handled better, along with the generic code elements. We performed a manual validation of the different pairing strategies on a sample code, containing all features of Java 8. The source and the results are available in the online appendix. The results of the large projects were also manually investigated.

In our final solution, we used the basic name-wise pairing for normal methods, line information-based pairing for anonymous methods, and a combined solution for generic methods. In this combined solution, if two methods have the same package, class and method name, have the same number of parameters, and have the same line information, it is assumed that they correspond to the same generic method declaration. The analyzers may represent the same generic method with different numbers of nodes in their call graphs. This asymmetry was solved by collecting every possible pairing between these nodes.

The manual validation proved that better pairing could be achieved if we could acquire more accurate line information of the methods. However, the reason for matchless nodes lies in the differences of the static call graph creators themselves, therefore, the matching of some nodes is impossible.

This research would not have been possible without Zoltán Ságodi, who fought the battles of the method name unification process in C++. From the joint evaluation of the results of each step, I was able to work out the next, heuristic step.

# 8 Systematic Comparison of Six Open-Source Java Call Graph Construction Tools

## 8.1 Overview

Our research on Java static call graph creator tools culminated in the work discussed in this chapter. We prepared for this study by developing the method pairing algorithm and our preliminary comparison [116]. We collected the six open-source static analyzer tools that were introduced in Chapter 6. For some tools, we had to extract the call graph from their internal representation, while others could be used immediately. In this work, our aim was to identify and analyze factors that could cause differences in the generated static call graphs in addition to the obvious algorithmic differences. For example, 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 [64], [96], [49], [75], [48], [65]. 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. This comparison may help to take the appropriate considerations into account when developing a call graph-based algorithm.

We conducted the research as follows. The tools were challenged on an example code containing the language features of Java 8 (it is available as an online appendix[1]). Based on this, we analyzed what could cause differences in Java call graph creation. We also evaluated the tools on multiple real-life open-source Java systems and performed a quantitative and qualitative assessment of the resulting graphs. The following Research Questions (RQs) guided the direction of our comparison:

- **RQ1:** How does the different handling of Java's language features affect the resulted call graphs?

[1]`http://www.inf.u-szeged.hu/~ferenc/papers/StaticJavaCallGraphs/`

- **RQ2:** How different could the call graphs be in practice?
- **RQ3:** Do we get the same graphs if we ignore the known differences?

The rest of this chapter is structured as follows. The study of the factors causing the differences can be found in Section 8.2. The comparison results of the Maven[2] and ArgoUML[3] projects are presented in Section 8.3. The results of the other projects are available as part of the online appendix. The threats to validity are examined in Section 8.4 before we draw our conclusions in Section 8.5.

## 8.2 Factors of Differences

In this section, we collect the factors that are responsible for the differences between the generated call graphs. This enumeration will help with answering the first RQ.

### 8.2.1 Initializer Methods

The handling of the different types of initializations is one of the main sources of difference. Naturally, all of the tools represent constructor calls. All but JDT detect and connect generated default constructors even without the instantiation of an object. Derived classes' calls to super constructors are represented as well. In case of source code based tools, initializer blocks and constructors are portrayed as two seperate 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 implicit parameter, since it is not present in the actual code. Both solutions are acceptable, and do not lessen the accuracy of the graphs, although it made the node pairing more challenging.

Static initializer blocks are executed when a class is loaded by the class loader of the Java Virtual Machine. As it was mentioned in Chapter 3, it is a dynamic process, triggered by different types of usage of a class. Representing this is a challenge, not only in symbolic execution, but also in call graph creation. The inclusion of such node in the call graph can be a little unpredictable and cumebersome. All tools represent static initializer blocks, however, in varying degrees of detail. A large part of Soot's graphs are made up of static initializer block nodes. When a class is used and it has at least one static field declared, Soot inserts a corresponding static initializer node.

### 8.2.2 Polymorphism

Polymorphism means that an object can take on many forms. The most common use of polymorphism in object-oriented languages occurs 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

[2]`https://github.com/apache/maven`
[3]`http://argouml.tigris.org/`

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) [96]. 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 [64], [96], [49], [75], [48], [65], 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 a method in a given inheritance hierarchy is callable at the polymorphic call sites. In many cases, this will result in some false-positive call edges, as we will see later in the discussion of anonymous classes. The `ZeroOneContainerCFA` algorithm of WALA is more sophisticated, but it is not always accurate either. Neither WALA nor Soot depicts method invocations of default methods of the interface classes. JCG also has a specific behavior. If a method is not overridden by the derived class, it generates a copy of the base method in the derived class. This method does not refer to the original one and the methods it calls are not connected either. If an application wants to traverse possible execution paths based on this graph (for instance a symbolic execution engine), it can skip some possible paths.

### 8.2.3 Anonymous Source Code Elements

As we have already described in connection with the development of the pairing algorithm, anonymous methods and classes are challenging to handle because there is no standard naming convention. If the tools do not provide valid line information, some anonymous methods will remain unmatched, which results in differing call edges as well. However, our examination revealed that anonymous elements can cause other important differences. As OSA, SPOON, and JCG do not employ specific algorithms to handle polymorphism, it is not surprising that they do not depict call edges to the methods overloaded by anonymous classes. Soot's CHA algorithm can cause discrepancies too, because it consider these methods to be a part of the class hierarchy. However, this assumption is not always correct, since anonymous classes' methods are not reachable in many contexts. This can result in false-positive call edges in the graph.

### 8.2.4 Generic Elements

As it was described in the previous chapter, generic source code elements are parameterizable classes and methods that can cause trouble in the node pairing mechanism. The Java compiler applies type erasure to generic elements. This means that the compiler enforces type constraints at compile time. It replaces all type parameters in generic types with their bounds or with `Object` if the type parameters are unbound. This type erasure is used by the call graph tools in most cases, but not all. Although WALA and Soot are 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. This also results in differences.

### 8.2.5 Java 8

Java 8 has brought several significant changes to the language that have also affected the construction of call graphs. It introduced the concept of functional interfaces. These are interface classes containing exactly one abstract method (and any number of default methods). The `ActionListener` interface is a good example of functional interfaces. Lambda expressions and method references, which are also new features of Java 8, can be used to represent the instance of a functional interface. Considering this, lambda expressions cannot be reckoned as methods, therefore, their interpretation in a call graph is a bit cumbersome. All tools but WALA represent lambdas in the graph by the interface they implement. WALA creates dedicated nodes for lambda expressions. It handles functional interfaces with specific nodes, to which the lambda and method reference nodes are connected. From our research, we concluded that most call graph builder tools fail to detect and, therefore, represent which actual methods are called through functional interfaces. Although it would be possible by tracing the inner calls of the Java libraries, they do not link the actual implementation to the call site

### 8.2.6 Dynamic Method Calls

Java is a dynamic language. This feature is not limited to polymorphic method calls, Java also supports reflection and the so called method handle mechanism. Reflection allows to manipulate a class and its members (field, methods etc.) at runtime. The method handle mechanism serves a similar purpose, but it implements it differently. We tested on our sample code whether any of the six tools could determine the targets of a basic reflection or 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. The result was similar in the case of native JNI calls and callback methods.

### 8.2.7 Answearing RQ1

The examination presented in the previous subsetions provides an answer to the first RQ. The different treatment of language features can cause significant differences in the generated graphs. Naturally, the handling of polymorphism can cause differences in edge numbers. Additional nodes may appear as well. Sometimes this is due to the byte code nature of the tool (e.g. default constructors or generated methods found by Soot, WALA, and JCG). Bytecode based tools might also represent extra, implicit paramteres not present in the source code itself. In other cases, the extra node is only a technical help for call graph construction (e.g. linking inherited methods by JCG).

## 8.3 Quantitative and Qualitative Analysis

In this section, we perform a quantitative and then a qualitative analysis of the differences in the graphs. The examination is presented on the results of the sample code and of the ArgoUML-0.35.1[4] and the Maven-3.6.0[5] systems. ArgoUML is a UML modeling tool with 180 KLOC, while Maven is a library tool with 80 KLOC, and older

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

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

versions of both are also presented in the Qualitas Corpus database [94]. Analyses were also performed on other systems. The measurements of these and the sample code are available as an online appendix at `http://www.inf.u-szeged.hu/~ferenc/papers/StaticJavaCallGraphs`.

### 8.3.1 Answearing RQ2

To answer RQ2, we need to perform a quantitative analysis of the differences in the graphs. Tables 8.1 and 8.2 summarize the results of the sample code. In case of Table 8.1 the numbers in the main diagonals are the number of methods found by the corresponding tool (e.g. Soot found 114 methods). Every other cell in a row shows how many percent of its methods was found by the tool in its column. The structure is similar for Table 8.2. The diagonal elements depict how many calls were detected by the tool (for example, Soot identified 404 invocations), while the other cells show the percentage of the coverage reached by the other tool in the column. For example, WALA found 249 calls and 175 of them were found by JDT as well, which results in 70.28%. JDT detected 211 call edges from which WALA found 175 as well. However, as JDT has fewer calls than WALA, the ratio is higher, 82.94%. That is why the table is not symmetrical. For an easier visual overview, the percentages above 80% are colored green, while the percentages below 60% are red.

The six tools together found 176 distinct methods and 472 different method invocations. These numbers are presented in the top left cell of the tables, respectively. The number of calls discovered by the individual tools ranges from 211 to 404.

The results of the sample code highlight the significant differences that can be experienced even with such a small input. We can discover similarities between the tools. One of the trends that can be seen even from the tables of the sample code is that the results of OSA and SPOON are well aligned. OSA covers all the methods and edges of SPOON's graph. SPOON connects three additional library methods into the graph, therefore, conversely, full coverage is not achieved.

| **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 8.1:** Common methods of the sample code

The trends and differences are more pronounced in the case of the large projects. Table 8.3 and Table 8.4 show the differences in the nodes of the Maven and the ArgoUML projects' callgraphs, respectively. The differences of the calls are shown in Table 8.5 (Maven) and Table 8.6 (ArgoUML).

It can be seen, that Soot represents considerably more methods in case of the Maven project. For the edges of the graphs, Soot finds overwehlimgly more than other tools. The reason for this is the detailed portrayal of static initializer nodes, and the representation of every overridden method which is caused by the CHA algorithm. On

| **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 8.2:** Calls of the sample code

| **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 8.3:** Methods of the Maven 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** |

**Table 8.4:** Methods of the ArgoUML project

the other hand, WALA contains fewer methods (and thus fewer edges) than all the other tools, thanks to its precise pointer analysis, and to the fact that it builds the call graphs from certain entry points. It remained unexposed in case of the sample code, but the results of the large projects show that OSA finds noticeably different methods compared to other tools. OSA performs a library based analysis, meaning that it analyzes every

| **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 8.5:** Calls of the Maven project

library that is on the given file path. It can only resolve library references based on names or the actual type only. OSA will not process library functions by searching on the projects' class path like other tools. Let us imagine that a `void foo(Object o)` method is declared in an unanalyzed directory. When OSA finds a `foo("string parameter")` call in the analyzed source, it will assume that the called method was declared as `void foo(String s)`. This causes nodes to appear in the graph that do not occur in the case of other tools. Since, in most cases the called methods do not even have line information, pairing cannot be achieved.

With RQ2, we examine how different the graphs can be in practice. The previously presented results show considerable differences. Although the pairing algorithm does not pair all possible nodes due to the lack of line information, the examples described above prove that the differences are not primarily due to this. Even on the sample code, one tool defines twice as many edges as the other (Table 8.2), but in case of the large projects, the number of edges differ even more considerably.

## 8.3.2 Answearing RQ3

RQ3 examines how similar the graphs will be by eliminating the known differences. Do we get the same resulting graph by each of the tools if we ignore the known differences? It is important to examine this issue. If we remove the parts in connection with the known differences, the differences that have gone unnoticed before can also become visible. This in-depth comparison can help identify the tools features.

Based on the factors identified in the previous section, there are three major attributes that influence the results of call graph creator tools. Here is a list of these aspects.

- **The handling of Java language elements.** Does the analyzer represent the constructor and the initializer blocks as one or two nodes? Are the lambda expressions represented by the interface or by a dedicated node? Further examples can be found in Section 8.2.
- **Processing differences.** Does the tool employ library based analysis or some kind of pointer or reachability analysis?
- **Algorithmic differences.** How does the tool deal with dynamic calls that cannot be resolved during static analysis? What kind of algorithm does it use to make the connected calls more accurate?

In the following subsections, we are going to remove the discrepancies caused by these aspects. In each step, we create a subgraph from the original with the help of

| **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** |

**Table 8.6:** Calls of the ArgoUML project

a 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 a removed node are eliminated as well. We will demonstrate how the graphs become more and more similar to each other.

| **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 8.7:** Common calls of the Maven project after eliminating the clinit calls detected only by Soot

| **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** |

**Table 8.8:** Common calls of the ArgoUML project after eliminating the clinit calls detected only by Soot

### Eliminating Differences Caused by Language Elements

Section 8.2 showed that certain language elements can significantly increase the amount of nodes, and, therefore, edges in the graph, which can cause large differences. For example, Soot represents every class's static initializer block time if it has at least one static member. Because of this, 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. The following two properties had to be met for filtering: the nodes only have incoming edges in Soot's output and do not appear in the other tools' graphs. Table 8.7 and 8.8 show the results of the removal of the Soot-only static initializer (clinit) nodes and corresponding edges. We only present those tables that contain the call edges, as in case of methods the changes in the tables are less striking. Naturally, the difference compared to Tables 8.5 and 8.6 is observable only in the first row as the removal only affected Soot's graph. In case of Maven, 62 nodes and 992 edges were eliminated, while in case of the ArgoUML project 498 static initializer methods and 12,770 corresponding edges were deleted from the graph. Based on the tables, we concluded that filtering Soot's static initializer nodes did not eliminate the differences significantly. Additional filtering steps were introduced.

| 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 8.9:** Common calls of the Maven project after eliminating library calls

| 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** |

**Table 8.10:** Common calls of the ArgoUML project after eliminating library calls

### Eliminating Algorithmic Differences

As we have seen in the previous subsection, it is not only the handling of Java language elements that causes significant differences in the graph. 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 represent methods called within library methods. Other tools provide more detailed information about calls outside the source of the input project. When developing a call graph-based application, we need to consider the depth of the dependency exploration that is necessary for the task. Is it important for us to examine the dependencies generated by the execution paths or is a less accurate representation sufficient? In certain cases, the library functions may call the project's methods through call back methods. With a less accurate representation, this information is lost. Call graph based applications may be sensitive to this.

Based on these observations, our next filtering step was to eliminate the library methods and the corresponding edges from the graph. Table 8.9 and 8.10 show the differences of the call edges after this modification. In most cases, the graphs became more similar, but there are exceptions. We filtered out edges that were detected by multiple tools, which reduced the similarity in those cases. The numbers show that a lot of edges have been removed from Soot's graph that the other tools did not detect. This means that Soot represents the library nodes with more detail.

### Eliminating Processing Differences

Finally, we examined the differences that come from the different processing mechanisms of the tools. There are many low percentages in WALA's column in Table 8.9 and 8.10. This is because the other tools represent many methods that WALA does not. WALA starts the call graph building algorithm only form selected starting points

| **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 8.11:** Common calls of the Maven project between methods recognized by all tools

| **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** |

**Table 8.12:** Common calls of the ArgoUML project between methods recognized by all tools

(e.g. the `main` method if the project has one) and detects the reachable nodes only. Other tools take all methods into account that are present in the code base, while some tools consider only those methods that are reachable from public methods. This diversity in the processing mechanism can cause a lot of differences, as quite different nodes will build up the graphs.

Seeing how much difference there was still left in the graphs, we decided to take a bold step. We only kept those methods in the graphs that were found by every tool. All other methods and their associated edges were filtered. We expected better comparability of the graphs from this experiment. For example, the tools that use the same analysis algorithm would give more similar results. As we can see in Tables 8.11 and 8.12, the results do not support this assumption. For instance, the results of SPOON, OSA, and JDT (which employ only static analysis) still differ.

In order to find the cause of the remaining differences between the edges, 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 even worse, because out of 9,477 edges there are 6,112 that are not found by every tool, which is 64%. Soot and WALA apply pointer analysis during graph construction, which explains most of the edges that are only discovered by these tools and the edges that are only found by the other, not pointer analysis based tools. However, there are 62 edges in the Maven project and 872 edges in the ArgoUML project, that cannot be explained by the various pointer analysis algorithms. Their examination revealed previously undiscovered factors. In the following paragraphs, we list these phenomena.

One such factor are the errors that are present in the graphs. For example, the tools might represent call edges that correspond to invalid call paths or execution orders. In case of SPOON and JDT, the initializations of the static blocks were 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 a default constructor (19 and 42 edges for Maven and ArgoUML, respectively).

In the case of SPOON, we discovered another misbehavior that occurred very rarely when analyzing the large systems. Let us assume that two classes from different packages but with the same name are present in the class path. Both classes have a method with the same signature (same return value, same parameters, same name), but, in reality, only one of the methods is called. SPOON will not distinguish the two classes if such method is invoked; it will create two call edges which is clearly a mistake. Besides this phenomenon, there were 2 cases in Maven where SPOON created call edges, although the called method in the graph had a different number of parameters than the caller provided in the source code. We found one erroneous edge in Maven's graph and two in ArgoUML, because OSA and SPOON handled an overloaded[6] method improperly, however, in other cases, no such error was found.

In certain cases, bytecode based analyzers (Soot, JCG, and WALA) placed loops in the call graph that do not exist in the actual code (7 cases in ArgoUML). This happened when a method was overridden, but the derived class changed the type of the return value. The compiler generates a node to handle the different return values, and the generated and the original methods are connected. However, methods cannot be distinguished based on return types, therefore, source code based analyzers represented them with the same node. 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. Compared to this, source code based analyzers represent this method with only one node which does not have a loop edge, because methods cannot be distinguished based on return types.

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

We found a special case, when an anonymous class was implemented in a parameter list of a method. There was a call within the method of the anonymous class. Soot handled this by creating 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 a 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.

With RQ3, we investigated whether the outputs of the call graph builder tools would be the same by eliminating the known differences. Our manual examination proved that the answer is no. Plenty of minor differences remained in the graphs that we would not have thought of at the beginning of the research. Depending on which features are more important to us in an application (e.g.: the precision of the control

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

flow information, or the dependencies defined between the methods), we must take the features of the call graph tools into account and choose the most appropriate for our purposes.

## 8.4 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 that could have fulfilled our selection criteria. In addition, the tools have many parameters that influence the construction of call graphs (e.g. different kinds of pointer analysis) but since we focused on the tools themselves instead of the differences between the algorithms, we executed each tool with only one configuration. Our goal was not to compare the call graph builder algorithms themselves, but to investigate how many factors may cause the output of the tools to differ. We examined the extent to which they are responsible for the discrepancies of the call graphs. Therefore, we were not looking for the optimal setting of a tool, instead we used one that reliably worked on the tested inputs.

In case of OSA, SPOON, and JDT, we implemented the call graph exporter ourselves. Although they were thoroughly tested with a sample code containing all Java 8 features, we could have made mistakes during the development. Naturally, there may always be errors if the data extraction is left to the user.

Further inaccuracies may be caused by not taking into account the configuration xml-s and the 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.

The method pairing mechanism that was described in the previous chapter can also be considered a threat to validity. As we pointed it out, it was not possible to rely solely on the methods' names. Because of the anonymous and generic elements, 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.

## 8.5 Conclusion

Java is a general programming language with many useful, dynamic features that is still very popular today. There are thousands of applications and websites that rely on Java. As a commonly used language, it is often subjected to static analysis, vulnerability checks, such as symbolic execution, and other inspections. One of the main pillars of static analysis is call graph creation, and although it might seem like a straightforward task, 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. However, we did not find any articles that would examine the differences from a practical perspective like we did.

We examined how 6 open-source call graph building tools perform when analyzing an artificial example and some larger-scale, open-source projects. These six tools can be considered a representative sample of the currently available, state-of-the-art call graph generators. Before the comparison, we collected the factors that could cause differences in the generated call graphs. For example, there may be differences in how the calling contexts are represented by the tools or in their way of processing. We evaluated the effect of these differences in the resulting call graphs on 4 open source systems. The results of Maven and ArgoUML systems were thoroughly discussed, while the the rest are available in as an online appendix. Through these examples, we have shown the extent to which each factor affects the dissimilarity of the call graphs. In this way, we also highlighted the parameters that can significantly determine the resulting call graphs.

We continue our research on Java call graphs by comparing the static call graphs to a dynamic call graphs. A dynamic call graph can be created by concretely executing the program, for example executing the test suite of a system with a profiler. Thus, a dynamic call graph can be exact, but it only contains those methods and calls that have actually been executed. It is an exciting question to examine what static graphs are capable of compared to a dynamic graph.

# 9
# Summary of Part II

The previous three chapters summarized my research on call graphs. Call graphs can be used as a basis for various static analysis techniques, so research of them contributes to the quality assurance of source code through static analysis. Therefore, our original goal was to examine the call graphs generated by 6 open-source static Java analyzers. This research can help developers find the appropriate call-graph creator tool for their purpose. However, we encountered difficulties at the beginning of the study, caused by the fact that the analyzers represented certain source code elements, e.g. constructors, initialization blocks, anonymous source code elements, and generic source code elements differently. The unification of method names was needed, in order to be able to match the corresponding nodes to each other. To do so, we developed a pairing algorithm with several heuristic steps. My main role was to design the heuristics and evaluate the results. The handling of anonymous and generic source code elements required the most attention. To deal with anonymous elements, we developed a method called anonymous transformation. As anonymous source code elements have a non-standardized name, we decided to replace the non-standardized part of their name with a constant string, and then perform a simple name-based pairing. If a class has multiple anonymous classes, this approach causes a loss in the accuracy of the pairing, therefore, to refine the accuracy, we introduced a line information-based pairing instead. However, this also caused challenges, so it was applied for anonymous and generic methods only. Lastly, we introduced a method to handle generic elements represented by multiple method signatures. The problem is that if a generic method is instantiated with different types, tool A might represent it with multiple nodes, while tool B may represent it with only one node. We solved this problem by collecting all possible pairings.

A manual validation of the heuristic pairing approach was performed, because, although the algorithm has significantly improved the graph pairing, many methods remained unmatched. We wanted to investigate if we could improve the pairing algorithm. However, our in-depth examination revealed that most of the unmatchings cannot be resolved. A better pairing could be achieved if we could acquire more accurate line information of the methods. However, in most cases, the discrepancy is caused by the differences of the static call graph creators themselves, therefore, the matching of numerous nodes is impossible.

In our next research, we performed an in-depth analysis of these remaining differences. We ran the six analyzer tools on a sample project and on real-life open-source Java systems as well and performed a quantitative and qualitative assessment of the resulting call graphs. We identified six sources for the differences: the handling of initializer methods, polymorphism, Java 8 language elements (such as lambdas), dynamic method calls, generic source code elements, and anonymous source code elements. The different treatment of these features causes significant differences in the generated graphs. In addition, a multi-step filtering method was completed. In its final stage, we kept only those methods in the graphs that were found by every tool. Nevertheless, the differences in graphs have not disappeared. Some of the discrepancies can be explained by various pointer analysis techniques, but not all. For example, in the case of ArgoUML project, out of the 9,477 edges that remained in WALA's graph, there are 872 edges that cannot be described that way. Their examination revealed previously undiscovered factors such as errors or features of bytecode-based analyzers.

In this thesis point, we implemented a heuristical call graph pairing algorithm, so we could reveal the extent and the causes of the differences between the call graphs generated by various static analyzer tools. We proved that the outputs of the call graph builder tools would not be the same even by eliminating the known differences.

# Part III

# Development and Evaluation of Primitive Obsession Metrics

# 10 Primitve Obsession: an Overlooked Code Smell

## 10.1 The Forgotten Smell

During development, the initial structure of the source code will degrade. It will become more difficult to identify the original design and understand the code, therefore, later changes and bug fixes will have higher costs. Refactorings, i.e, modifications that improve different attributes (e.g., readability, complexity, or maintainability) without changing the external functionality [39], should be applied regularly to preserve the quality of the code.

The 22 bad code smells collected by Fowler and Beck [39] can be interpreted as indicators that suggest that some refactoring is needed in the code. Code smells are not actual coding errors, but may indicate deeper design problems that may reduce the maintainability of the code in the future [73]. It is therefore not surprising that a considerable amount of research has been conducted on the importance of code smells and their impact on maintenance costs. Through the years, in addition to the original 22 code smells, several new types have been introduced and various detection techniques have been proposed to help developers identify and refactor problems in the code.

Some code smells were less studied than others. Primitive obsession, which roughly translates to the overuse of primitive types, is such a marginalized code smell. Previously, only one automated detection method was introduced for it in the thesis of Roperia [84], however, as a code smell, it can be a useful indicator of underlying design flaws. Therefore, we decided to study Primitive Obsession and defined several indicators for Java that can highlight the potential places where this bad smell could occur. We proposed several metrics based on these indicators. We integrated the metrics into a static analyzer tool and performed multiple experiments and studies.

The rest of this chapter focuses on the theoretical considerations and suggested metrics. Section 10.2 provides related work on code smells. Section 10.3 describes Primitive Obsession through an example and examines the difficulties in its definition. Section 10.4 introduces the Primitive Enthusiasm-based Primitive Obsession metrics, while Section 10.5, 10.6, and 10.7 introduce the metrics that I constructed for Primitive Obsession detection.

## 10.2 Related Work

There are numerous studies about the impact of code smells on code quality. This section summarizes some important works.

Many papers discuss how code smells are good indicators of maintenance problems [106], [73]. Although they are not the universal remedy for defect detection, they can provide valuable insights into some important maintainability factors, especially when more of them are combined. Moonen and Yamashita demonstrated in an empirical study [73] that some of the code smell definitions are useful to evaluate the maintainability of a project. They listed the factors that are important from the software maintainers' perspective and identified which smells are related to them. Expert advice was used for this evaluation. They found that some of the current code smell definitions (such as God Class, Long Parameter List, God Method, etc.) can be used for evaluating the maintainability. Primitive Obsession is not mentioned in this study, but there was no exact definition for it at that time. In another study [105], Moonen and Yamashita investigated how the interactions of multiple smells affect maintainability. They automatically detected 12 code smells in several systems using Borland Together and InCode. Primitive Obsession was not among these smells. Principal Component Analysis (PCA) was used to detect co-located code smells. They found that certain inter-smell relations were associated with maintenance problems. Based on developers' opinions, they identified several artefacts that need to be prioritized during refactoring. Khomh et al. [60] found a relation between code smells and class change-proneness, which can support quality assurance and focus testing activities. In contrast to these studies, Sjøberg et al. investigated the connection between code smells and maintenance effort [90], and found that the work of changing code with smells was not significantly higher than the work of changing code without smells. The 12 examined smells have a limited impact on maintenance effort in contrast to code size and the number of changes.

Seeing how useful code smells can be, it seemed to be a remarkable shortcoming that Primitive Obsession lacked the interest of the research community. Zhang and his team analyzed the state-of-the-art knowledge about code smells in a systematic literature review [108]. They collected 319 papers from 2000 to 2009 and examined 39 in detail. In their first research question, they investigated which code smells attracted the most research attention. The results showed that Duplicated Code smell was discussed the most – in 21 papers out of 39 – whilst many bad code smells received very little attention. Primitive Obsession was among the unpopular smells having only 5 corresponding papers, although these papers examined all 22 of Fowler's bad smells. This indicates that Primitive Obsession was not studied individually in the investigated period. A more recent systematic literature review [50] came to a similar conclusion. Gupta et al. examined 60 research papers between 1999 and 2016 and found that four of Fowler's bad smells – including Primitive Obsession – had no detection method in any of the papers. A study on five known tools which could detect code smells as well [38] reported that none of them were capable of finding Primitive Obsession.

Mäntylä et al. carried out an empirical study on the subjective smell evaluations of developers [71]. They found that human factors (knowledge, work experience, role in the project) have an impact on the identification of a code part as a bad smell. Moreover, they compared the results of the subjective evaluations and metric-based detection techniques for three smells, and found that the metrics and smell evaluations did not always correlate. One of the smells they used in the comparison was the Long Parameter List, which had the largest correlation value. They also observed that the Long Parameter List smell mainly consisted of primitives, which could indicate a connection with the Primitive Obsession smell. We utilized this information during the creation of our metrics.

## 10.3 Introduction to Primitive Obsession

In most programming languages there are two categories of data types: primitive and complex. Primitive types are the most basic data types provided by a language, e.g. *boolean*, *integer*, *char* etc. Complex types like *classes* and *structs* are the composites of other existing data types e.g. primitive types and complex types. Complex types usually also include some semantic knowledge about the data they represent. They help developers to implement encapsulation, which is one of the major principles of object-oriented programming. For example, it is a convenient and more readable solution to place three integers that represent a 3D point in a class instead of using them separately.

Primitive Obsession means the overuse of primitive data types, and it is a symptom of the existence of overgrown, chaotic code parts. Instead of creating small objects for small tasks, the programmer scatters the data among primitive types, making the code less readable. According to the Mantyla taxonomy [70] it is a bloater type of smell.

Its definition can be broadly interpreted, and not really practical if we are considering automated detection. Our first step towards automatic detection was to characterize Primitive Obsession with exact, quantifiable descriptions. Chapter 3 of the Fowler book [39] yielded some pointers for us by providing a list of possible refactorings for Primitive Obsession. A small GitHub project[1] created by Steven A. Lowe also aided our understanding by showing these refactorings step-by-step in practice. The following paragraphs depict the many faces of this smell through the example in Listing 10.1.

The example shows a schematic class. Its constants `ENGINEER` and `SALESMAN` at Lines 3 and 4 can be considered type codes. Type codes are a set of integer or string variables that usually have an understandable name, and they are employed to simulate types, e.g. different types of employees. Although they are widely used in projects, they are a kind of Primitive Obsession, as they violate the object oriented paradigm and can cause *hidden dependencies* [107]. Type codes can be removed by forming a class or, for example, with a *State* or *Strategy* pattern [41].

```
1   class Employee{
2
3     static const int ENGINEER = 1;
4     static const int SALESMAN = 2;
5
6     public void workOffice(int from, int to, int numberOfBreaks,↩
          String task){
7
8       if(task == null
9          || task.length() == 0) {
10        /*...*/
```

[1]https://github.com/stevenalowe/kata-2-tinytypes

```
11        }
12
13        switch(type){
14          case ENGINEER: /*...*/
15          case SALESMAN: /*...*/
16        }
17      }
18
19      public void workHome(int from, int to, int numberOfBreaks, ↩
           String task){
20        if(task == null
21           || task.length() == 0) {
22          /*...*/
23        }
24
25        switch(type){
26          case ENGINEER: /*...*/
27          case SALESMAN: /*...*/
28        }
29      }
30    }
31
32    class Workplace{
33      public Employee worker;
34
35      public Workplace(){
36        worker = new Employee(Employee.ENGINEER);
37      }
38
39      public void work(int from, int to, int numberOfBreaks, ↩
           String task){
40        if(isPandemic()){
41          worker.workHome(from, to, numberOfBreaks, task);
42        }
43        else{
44          worker.workOffice(from, to, numberOfBreaks, task);
45        }
46      }
47    }
```

**Listing 10.1:** Sample code containing multiple Primitive Obsessions

At Line 19, the parameter list of the `work` function is described. Three of its parameters are integers, whilst the fourth is a string. Parameter lists like this are a kind of Primitive Obsession, especially if they appear several times in the code. (Strictly speaking, strings are not primitive types, but logically it is practical to include them in the definition.) They can be refactored by introducing a parameter object. Lines 20 and 21 also confirm the need for refactoring, because value checks like this should usually be encapsulated.

Listing 10.2 shows the previous example after refactoring. To save space, irrelevant constructors and methods are not present. There are parameter objects (`WorkShift`) on Line 18 and Line 20 instead of numerous primitive parameters. The type codes are replaced by a class hierarchy. The `Employee` and `Salesman` subclasses provide the specialized behaviour which was previously implemented with switch-case statements. The value check was also eliminated. There is a method call (`isEmpty()` on Line 6) instead and it is also encapsulated in a function.

```java
class WorkShift{
  private int from, to, numberOfBreaks;
  private String task;

  public boolean isValid(){
    return task != null && task.isEmpty();
  }
}

abstract class Employee {

 public void checkShift(WorkShift s){
   if(!s.isValid()){
      /*...*/
   }
 }

 abstract public void workOffice(WorkShift s);

 abstract public void workHome(WorkShift s);

}

class Engineer extends Employee{
 public void workOffice(WorkShift s)
 {
    checkShift(s);
    /*...*/
 }

 public void workHome(WorkShift s)
 {
    checkShift(s);
    /*...*/
 }
}

class Salesman extends Employee{
 public void workOffice(WorkShift s)
 {
    checkShift(s);
    /*...*/
 }

 public void workHome(WorkShift s)
 {
    checkShift(s);
    /*...*/
 }
}

class Workplace{
  public Employee worker;

  public Workplace(){
    worker = new Engineer();
  }

  public void work(int from, int to, int numberOfBreaks, ↩
      String task){
    WorkShift shift = createWorkShift(from, to, numberOfBreaks↩
        , task);
    if(isPandemic()){
      worker.workHome(shift);
    }
    else{
      worker.workOffice(shift);
    }
  }
}
```

**Listing 10.2:** The refactored version of the example code

More examples are available in the previously mentioned GitHub project.

### 10.3.1 Challenges in Definition

As was highlighted in the previous example, it is challenging to give Primitive Obsession an exact, quantifiable definition. It has various aspects that can hardly be described with a single formula. The excessive use of primitive types can appear in parameter lists or in the form of type codes, etc.

Since defining Primitive Obsession is such a complex issue, we deconstructed the smell and addressed its characteristics with multiple formulas. We created six metrics to highlight Primitive Obsession-infected code parts [112], [114]. Three of the six metrics are my own work. These will be presented in detail in Sections 10.5 - 10.6. The other three metrics will be referred to as Primitive Enthusiasm-based (PE) metrics. I was not the one who worked out the ideas behind these, but they were used in the measurement of Chapter 11, therefore, they are introduced in Section 10.4. Collectively, we will refer to the six metrics as PO metrics. They can be used as an indicator to highlight potential candidates for Primitive Obsession.

## 10.4 Background

### 10.4.1 Primitive Enthusiasm

Primitive Enthusiasm is a method level metric that captures and reports one important aspect of the Primitive Obsession bad smell [114]. It helps detect the overuse of primitive types in method parameters. It is based on Formulas 10.1 and 10.2. The definitions of the parameters are the following:

- $\boldsymbol{PrimitiveTypes}$ is the set of types that are handled as primitive ones.
- $\boldsymbol{N}$ represents the number of methods in the current class.
- $\boldsymbol{M_i}$ denotes the $\boldsymbol{i}$th method of the current class.
- $\boldsymbol{M_c}$ denotes the current method under investigation in the current class.
- $\boldsymbol{P_{M_i}}$ denotes the list of types used for parameters in the $\boldsymbol{M_i}$ method.

- $\boldsymbol{P_{M_i,j}}$ defines the type of the $\boldsymbol{j}$th parameter in the $\boldsymbol{M_i}$ method.

$$\begin{aligned} Primitives(M_i) := \langle P_{M_i,j} | 1 \leq j \leq |P_{M_i}| \\ \wedge P_{M_i,j} \in PrimitiveTypes \rangle \end{aligned} \tag{10.1}$$

Formula 10.1 describes how the primitive-typed parameters are collected for a given $M_i$ method. We select all the parameters from the $M_i$ method that can be found in the previously introduced $PrimitiveTypes$ set and return them in a list.

The calculation of Primitive Enthusiasm is illustrated in Formula 10.2.

$$PE(M_c) := \frac{\sum\limits_{i=1}^{N} |Primitives(M_i)|}{\sum\limits_{i=1}^{N} | P_{M_i} |} < \frac{|Primitives(M_c)|}{| P_{M_c} |} \tag{10.2}$$

The left-hand side of the inequality in Formula 10.2 expresses the percentage of how many of the parameters of the current class are primitive types. Sum the number of primitive types in the parameter list for each method in the currently processed class and divide this value with the total number of parameters in the class methods. The right-hand side of the inequality in Formula 10.2 calculates the percentage of the primitive types used in the currently investigated method. The number of primitive types in the current method is divided by the total number of parameters. Both sides of the inequality will calculate a number between 0.0 and 1.0 as the number of all parameters is always greater than or equal to the number of parameters that are primitive types. If the inequality is satisfied for a method, we mark it as possible Primitive Obsession. It is important to note, that only methods with at least one parameter can be used as the subject for the Formula. Methods without parameters do not use any primitive types as input, thus they are not the target of Primitive Obsession code smell.

### 10.4.2 Local Primitive Enthusiasm

Primitive Enthusiasm is defined as the function of the current and the overall primitive type usage in the method parameter lists of a class. This was the first PO metric we defined. Based on our initial evaluations, we realized that it needed to be refined to capture more aspects of Primitive Obsession. Multiple variants were designed which are intruduced in the following subsections.

As a first step, we renamed the Primitive Enthusiasm metric to Local Primitive Enthusiasm (LPE). The formula is calculated method-by-method in a given class context, therefore, it is considered as a local value. The value of LPE for a given $M_c$ method ($LPE(M_c)$) is either true or false depending on the satisfiability of the inequality in Formula 10.2.

### 10.4.3 Global Primitive Enthusiasm

Global Primitive Enthusiasm (GPE) is shown in Formula 10.3, where $G$ is a list which contains all the methods in the analyzed application. The right-hand side of the inequality is the same as in Formula 10.2, the difference can be seen on the left-hand side. Unlike in LPE, the global primitiveness ratio is calculated based on every method in the project and the examined method is compared to it.

$$GPE(M_c) := \frac{\sum_{i=1}^{|G|} |Primitives(G_i)|}{\sum_{i=1}^{|G|} \mid P_{G_i} \mid} < \frac{|Primitives(M_c)|}{\mid P_{M_c} \mid} \tag{10.3}$$

The idea behind GPE is that it can be profitable to compare classes with each other to see which ones are outstanding with regard to the primitive type usage in the parameter lists of their methods.

### 10.4.4 Hot Primitive Enthusiasm

Both LPE and GPE report a set of methods that might be affected by Primitive Obsession. To focus the attention of the developer on the more suspicious code parts, the combination of LPE and GPE is proposed. The combined formula, Hot Primitive Enthusiasm (HPE) is presented in Formula 10.4. HPE only reports the methods that are reported by both LPE and GPE.

$$HPE(M_c) := LPE(M_c) \land GPE(M_c) \tag{10.4}$$

### 10.4.5 Result Aggregation

The PE variants in their current form characterize the project with a list of methods. These are the methods for which the formulated inequality in Subsection 10.4.1 and its variants were true. However, for large projects the list of reported methods can be long, making the review a demanding task. Therefore, we proposed a class level aggregation of the results. This way, the classes can be ordered by the number of reported methods.

## 10.5 Method Parameter Clones

The Method Parameter Clones (MPC) metric is a class level metric. After examining the nature of Primitive Obsession, it is reasonable to assume that if a class suffers from Primitive Obsession, certain method parameters will appear multiple times in the parameter lists of its methods. They will have the same type and – usually – the same name because logically they correspond to the same data. These are the parameters that could be extracted to a value object instead of using them separately. To grasp this property, the MPC metric was presented, which performs the following steps for each class in a project:

1. Initially the MPC value for a class is set to zero.
2. For each method parameter in the class, create a $(type, name)$ pair.
3. Select only the pairs where the type is in the *PrimitiveTypes* set.
4. Increment MPC value by one for every $(type, name)$ pair that appears at least three times.

The reason that only three or more repetitions are counted is *The Rule of Three* [39]. This means that two instances of similar code don't require refactoring, but when similar code is used three times, it needs to be corrected.

## 10.6 Static Final Primitive Usage

One more class level metric is the Static Final Primitives (SFP) metric. It captures another important aspect of the Primitive Obsession bad smell. The SFP metric measures the usage of class constant values as type codes. To check if a variable can be a candidate for further investigation, the Static Final Primitive function is used, which is described in Formula 10.5.

$$\begin{aligned} SFP(V) \;:=\; & isClassLevel(V) \\ & \wedge isStatic(V) \\ & \wedge isFinal(V) \\ & \wedge isUpperCase(V) \\ & \wedge typeOf(V) \in PrimitiveTypes \end{aligned} \tag{10.5}$$

The SFP function will return true for a variable ($V$) if it has all the following properties, thus it might serve as a type code:

- $V$ is a class level variable,
- $V$ is static (e.g. has a `static` modifier in Java),
- $V$ can be assigned only once (e.g. has a `final` modifier in Java),
- the name of $V$ contains only upper case characters, numbers, and underscores,
- and the type of $V$ is included in the previously defined *PrimitiveTypes* set (i.e. primitive).

Otherwise SFP returns false. With this function it is possible to filter out static final primitive variable usages in methods.

The Static Final Primitive Usage (SFPU) function is defined in Formula 10.6. It has three parameters: $M_c$ and $V$ denote the function and variable under investigation, respectively, while $F$ is a filter function. In the formula, $U_V$ denotes one usage of variable $V$. The $U_V \in M_c$ relation means that the $U_V$ variable usage (access or modification) is performed in the $M_c$ method. By applying the $F$ filter function it is possible to select a subset of the $U_V$ variable usages. The $SFPU$ function collects the access and modification statements for variable $V$ in the $M_c$ method, for which both the $SFP$ function and the supplied $F$ filter function return true.

$$SFPU(M_c, V, F) \;:=\! \left\{ U_V \;\middle|\; U_V \in M_c \wedge SFP(V) \wedge F(U_V) \right\} \tag{10.6}$$

## 10.7 Static Final Primitives - Switch Case Usage

A concrete application of the $SFPU$ function is the calculation of the number of static final primitive variables which are used as case labels in `switch` statements. This can be achieved by defining an $F$ filter function for the $SFPU$ that determines whether a given $U_V$ variable usage is a case label. By using the $SFPU$ and the given $F$ filter function a class level metric is constructed, which is named Static Final Primitive - Switch Case Usage (SFP-SCU). The calculation is done for each class to see how many times its $SFP$ variables appear as case labels globally in the project. This heuristical approach is based on the idea that type code variables most probably will appear in branching structures, especially in `switch-case` statements.

# 11

# An evaluation of the Primitive Obsession metrics

## 11.1 Overview

This chapter summarizes the evaluation of the PO metrics that I developed: the $MPC$, the $SFP$, and the $SFP - SCU$ metrics. This work was originally described in our 2018 article [114]. There was a preliminary study [112] for this research in which we developed Primitive Enthusiasm, the first PO metric. We implemented its formula in a static analyzer for Java. It was tested on the Titan[1] and the ArgoUML[2] real-life Java systems and on the already mentioned sample program seeded with Primitive Obsession[3]. Manual validation of the results was performed. The sample project has six branches according to the degree of refactoring. The first branch has the most examples for Primitive Obsession, for example, its five functions with long, primitive typed parameter lists. Our algorithm reported all five functions as potential methods suffering from Primitive Obsession. These five functions are indeed refactored in subsequent versions to use different types instead of primitive types. In case of the large projects, there are multiple reports of Primitive Enthusiasm. Based on our manual validation of the results it can be stated that the class with the most reported methods is indeed suspicious of Primitive Obsession.

To better cover the aspects of Primitive Obsession, we created the metrics listed in Chapter 10. The Primitive Enthusiasm metric was renamed Local Primitive Enthusiasm (LPE). We also reimplemented our initial static analyzer tool and evaluated it on three other Java systems. Again, we performed the manual validation of the results.

The chapter is structured as follows. We present the implementational steps, the analyzed systems, and our additional considerations in Section 11.2. Section 11.3 discusses the results of the manual validation, while the conclusions are placed in Section 11.4.

[1]https://github.com/thinkaurelius/titan
[2]https://github.com/stcarrez/argouml
[3]https://github.com/stevenalowe/kata-2-tinytypes

## 11.2 Research Setup

In this section, we report on the implementation of the PO metrics. We encountered two issues during implementation, for which we sought heuristic solutions. The analyzed Java systems are also characterized.

### 11.2.1 Implementation

The formulas were implemented in the OpenStaticAnalyzer (OSA) [5] for Java. OSA is an open-source, multi-language, static code analyzer framework developed at the Department of Software Engineering, University of Szeged. It is the basis for the SourceMeter system that was used in Chapter 2. The two analysis systems are similar in structure and operation, but the SourceMeter provides other functionalities as well, e.g. symbolic execution.

### 11.2.2 Determining Primitive Types

As the first step of the implementation, it was important to determine which types we were going to treat as primitives. Java contains the following primitive types: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`. Although `String` is by definition not a primitive type we decided to consider it as one. This established the set of primitive types, but it should be noted that Java has a wrapper class for each primitive type, such as `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Character`, `Float`, or `Double`. Although these types are classes, as my co-author, Péter Gál suggested, we decided to experiment with treating them as primitive types. Based on this, we formulated the following additional metrics: $\text{LPE}_{\text{W+}}$, $\text{GPE}_{\text{W+}}$, $\text{HPE}_{\text{W+}}$. These metrics are different from the originals in that the boxing types are also considered primitives. $\text{HPE}_{\text{W+}}$ is the intersection of the sets returned by $\text{LPE}_{\text{W+}}$ and $\text{GPE}_{\text{W+}}$.

### 11.2.3 Analyzed Projects

This subsection provides some detailes about the analyzed systems. We collected the following open source, real-life Java systems: *joda-time-2.9.9*[4], *log4j*[5], *commons-math-3.6.1*[6]. Table 11.1 summarizes their major properties to get an overall image about them.

| Project | KLOC | NC | NM | LPL |
|---|---|---|---|---|
| log4j | 16 | 189 | 1561 | 9 |
| joda-time | 28 | 249 | 4265 | 10 |
| commons-math | 100 | 1033 | 8808 | 14 |

**Table 11.1:** Properties of the examined projects: thousand lines of code (KLOC), number of classes (NC), number of methods (NM), longest parameter list (LPL)

[4]https://github.com/JodaOrg/joda-time
[5]https://github.com/apache/log4j
[6]https://github.com/apache/commons-math

### 11.2.4 Exclusion Strategies

During the implementation we decided to eliminate several methods from the examination. Constructor methods are skipped by default, as it is normal for them to have several primitive types in their parameter lists. The formulas for Primitive Enthusiasm calculation do not take into account that some methods only have one parameter or none at all. It is questionable to involve one-parametered methods in the measurement. Based on these considerations, we established two elimination strategies:

- skip every method with just one parameter (*Skip Ones* - *SO* strategy)
- skip only setter methods (*Skip Setter Methods* - *SSM* strategy)

Table 11.2 sums up the details of the two strategies for the examined three systems.

| Project | SSM | | SO | |
|---|---|---|---|---|
| | NNM | AVG | NNM | AVG |
| log4j | 1371 | 1.33 | 429 | 3.01 |
| joda-time | 3787 | 0.98 | 893 | 2.55 |
| commons-math | 7229 | 1.12 | 1953 | 2.93 |

**Table 11.2:** Comparison of the two elimination stategies: number of not eliminated methods (NNM), average parameter list length of not eliminated methods (AVG)

The numbers show that with the SO strategy only a subset of the methods was processed for the calculation. During the manual validation it was observed that the SSM strategy causes noise in the results. It is not surprising as it is hard to interpret Primitive Enthusiasm metric on methods with only one parameter.

## 11.3 Results by Metric

Primitive Obsession was first examined using the Primitive Enthusiasm metric [112], which was further developed in our study in 2018 [114]. These metrics suggest methods for refactoring that use an unusually large amount of primitive parameters. The study [114] showed that the Primitive Enthusiasm metrics report a high number of methods and classes. The metrics I have introduced can help highlight the classes that are most important to examine. The results are discussed in subsections 11.3.1 and 11.3.2. For the sake of completeness, Table 11.3 summarizes the aggregated (i.e. class-level) results of the Primitive Enthusiasm metrics. Compared to the Number of Classes (NC) column in Table 11.1 the number of reported classes for each PE variant is relatively high. Since our study found that the $\mathrm{HPE}_{\mathrm{W}+}$ metric provided the most accurate results, I used its results for the evaluation of my own metrics. The results of Table 11.3 were calculated using the SO exclusion strategy and the wrapper classes were included in the *PrimitiveTypes* set.

To validate the results of the metrics, the given warnings were processed manually. The reported classes were checked in the source code. As both the evaluation, and the judgment of Primitive Obsession themselves are subjective, deciding if a warning is true positive or not is difficult, therefore, the significance of the warning was investigated.

| Project | $LPE_{W+}$ | $GPE_{W+}$ | $HPE_{W+}$ |
|---|---|---|---|
| log4j | 165 | 217 | 153 |
| joda-time | 301 | 431 | 231 |
| commons-math | 698 | 1192 | 553 |

**Table 11.3:** The number of reported classes using the SO exclusion strategy

### 11.3.1 MPC Metric

The main interest in the MPC metric was not only to study its ability to highlight PO-infested classes but to examine how well the reported classes align with the classes detected by the aggregated Primitive Enthusiasm metrics. For the evaluation of the MPC metric, the SO exclusion strategy was used, and the wrapper classes were included in the *PrimitiveTypes* set.

The results of the MPC metric are summarized in Table 11.4. It reported 6, 50, and 90 classes on *log4j*, *joda-time*, and *commons-math*, respectively. The highest MPC count was 11 for *log4j*, meaning that in the `logMF` and `logSF` classes there are 11 parameter type - parameter name pairs that appeared at least three times in different parameter lists. Both of these classes were the top two results of the aggregated Primitive Enthusiasm metrics. They contain numerous methods that have a similar signature with multiple primitive typed parameters, so the metrics report a repetitiveness in the code that sometimes indicates a design flaw in the program. The `logXF` class, the base for `logMF` and `logSF` is also reported with an MPC count of 2 and has a high rank in the aggregated Primitive Enthusiasm results. The other three classes were also reported by at least one of the Primitive Enthusiasm variants.

The findings are similar for the other two projects as well. The top ten classes in the aggregated $HPE_{W+}$ results could be found in the MPC result set as well. Classes that have many methods with above-average primitive typed parameters usually have a lot of method parameter clones too. Therefore, the MPC metric can be a candidate for weighting the results of the aggregated Primitive Enthusiasm metrics and highlighting more repetitive parts in the code.

As the third column of Table 11.4 demonstrates, the intersection of the aggregated $HPE_{W+}$, and MPC result sets for *log4j*, *joda-time*, and *commons-math* has the following size: 5, 31, and 56. The numbers show that by intersecting the result sets, a significant number of warnings were pruned from both metrics. Although we may lose some useful warnings this way, we eliminate many noisy results.

| Project | MPC | $MPC \cap HPE_{W+}$ |
|---|---|---|
| log4j | 6 | 5 |
| joda-time | 5 | 31 |
| commons-math | 90 | 56 |

**Table 11.4:** Results of MPC. From left to right: the number of classes reported by the MPC metric, the number of classes in the intersection of the aggregated $HPE_{W+}$ and MPC results

### 11.3.2 SFP-SCU Metric

The metric was designed to detect type codes which were introduced in Section 10.3. It follows from the nature of type codes that they appear in conditional statements. The heuristical approach was to spot them through their usage in `switch-case` statements. Table 11.5 shows the results of SFP-SCU. It is important to note that during the evaluation no exclusion strategy was used, thus all methods were considered. However, the effects of the SO exclusion strategy combined with excluding the constructors were also investigated, and the results were the same for the studied projects. The wrapper classes were included in the *PrimitiveTypes* set.

The number of classes reported by SFP-SCU is in the second column of Table 11.5. These classes have primitive-typed static final class members that are suspected to be type codes. The size of the intersection between the results of the aggregated $\mathrm{HPE}_{\mathrm{W+}}$ and SFP-SCU is presented in the third column, whilst the size of the intersection with the MPC metric is in the fourth column. As this metric grasps a different aspect of Primitive Obsession than the Primitive Enthusiasm metrics, no significant overlap was expected.

| Project | SFP-SCU | SFP-SCU $\cap$ $\mathrm{HPE}_{\mathrm{W+}}$ | SFP-SCU $\cap$ MPC |
|---|---|---|---|
| log4j | 8 | 5 | 1 |
| joda-time | 13 | 4 | 2 |
| commons-math | 1 | 0 | 0 |

**Table 11.5:** Results of SFP-SCU. From left to right: the number of classes reported by the SFP-SCU metric, the number of classes in the intersection of the aggregated $\mathrm{HPE}_{\mathrm{W+}}$ and SFP-SCU results, the number of classes in the intersection of the MPC and SFP-SCU results

The constant variables, which can be found in the reported class of the *commons-math* project are used to distinguish between random number generation strategies. They only have one `switch-case` occurrence and could be replaced, for example, with an `enum`. The *joda-time* relies more on static final variables than the other two projects. These variables are members of multiple classes with the same name, making it harder to understand the code. A generalized solution could be considered instead of the scattered constant usage. In `log4j` a typical example for type codes can be found in the `PatternParser` class, which also appears in both intersections.

Though no hidden dependency check was done to study the seriousness of the possible type codes, it might be rewarding to refactor them for a more object oriented and readable solution.

## 11.4 Conclusion

In this research, we introduced and tested several Primitive Obsession metrics. The metrics were implemented in a Java static analyzer, evaluated on three large systems, and the results were analyzed. We experimented with two method elimination techniques and also considered including the Java wrapper classes in the primitive types' set.

The original PO metrics, the Primitive Enthusiasm variants can find methods that use more primitive types in their parameter lists than the average. It is not just a readability issue, but can be a sign for other bloater-type smells as well. However, to detect Primitive Obsession, it is necessary to represent its other aspects. Therefore, additional metrics have been introduced. The SFP-SCU metric is useful for typed code detection. In the future, we would like to consider whether or not a static final variable can be seen outside its class by giving the usages outside its class or package a different weight than the inner usages. The MPC metric reports classes that have repetitive, therefore, possibly smelly method signatures. The metric could be refined with ordinal information among the parameter clones.

The findings showed that the new metrics can highlight many smelly and hardly readable code segments. In the future, we would like to continue the study of these metrics and their combinations. The inclusion of enum constants in the *PrimitiveTypes* set could be an interesting experiment. Creating a Primitive Obsession benchmark is also a goal to provide a more objective comparison of the metrics.

# 12 Examining the Bug Prediction Capabilities of Primitive Obsession Metrics

## 12.1 Overview

Finding and fixing bugs is a very important task during software development to ensure the quality of the product. Software defects are very costly. The later they are discovered, the more expensive they are [20], therefore, there is considerable research interest in making the bug detection process more automated and efficient. Software fault prediction or, in other words, bug prediction is a possible approach to this effort. Considering this, it is no surprise, that there is a huge amount of literature on refining bug prediction [101], [110], [103]. This research also aims to find improvements with the help of three Primitive Obsession metrics [114].

There are six Primitive Obsession metrics introduced, but I only used three of them for this research: the $MPC$, the $SFP$, and the $SFP - SCU$ metrics. Hereinafter, the abbreviation 'PO metric' will only apply to these. The reason for their selection is that these metrics were originally defined as class-level metrics, and the source code metrics based bug dataset [37] that was expanded does not have method level records. After the expansion of said bug dataset, we studied the effectiveness of the prediction built upon it. We compared the new models with the results of the original dataset. While the cross-validation showed no significant change, in the case of the cross-project validation, we have found that the amount of improvement exceeded the amount of deterioration by 5%. Furthermore, the variance added to the dataset was confirmed by correlation and PCA calculations.

This chapter is organized as follows. Section 12.3 provides the background and the related work. The three research questions and their answers are presented in Section 12.3. Section 12.4 describes the threats to validity. We conclude our work in Section 12.5.

## 12.2 Background

Besides code smells and Primitive Obsession, our research covers two important domains: bug prediction and PCA calculation. The following subsections introduce these important fields.

### 12.2.1 Bug Prediction and Bug Datasets

Bug prediction is a method that supports the quality assurance during software development. Learning from the bugs of the past it builds a prediction model to foretell the location and amount of future bugs. It is a cost-effective approach compared to software testing and reviews. Menzies et al. report in their study [72] that the probability of detecting a bug by fault prediction may be higher than detecting it by the review process currently used in industrial environments.

Bug prediction can be used to estimate the number of remaining bugs in the code base or to identify the bug-prone parts of the code. Another possible use is to mine software defect associations [80]. In defect association, we define association rules, for example, if type A and type B defects both occur in the code, then a type C defect is likely to occur as well. This research examines the type of bug prediction that helps identify potentially buggy code parts.

Bug prediction can be based on statistical, expert-driven, or machine learning models [25]. This research focuses on machine learning based bug prediction. We used a unified bug dataset, which was introduced by Ferenc et al. [37]. Bug datasets contain historical data, such as previous defects, process metrics, or source code metrics [51], on which the prediction data is built. The dataset of Ferenc contains data from five publicly available bug datasets in a unified format: the PROMISE dataset [56], the Eclipse Bug Dataset [111], the Bug Prediction Dataset [30], the Bugcatchers Bug Dataset [52], and the GitHub Bug Dataset [97]. They contain class and/or file level source code metrics and information about the previous defects. We only utilized the PROMISE, the GitHub, and the Bug Prediction datasets, as only these contain class-level metrics. There are 66 project in the three datasets, the smallest only contains 6 classes while the largest contains 5908. To reduce noise, we decided to exclude those systems that contain less than a 100 classes. Thus, a total of 58 systems remained, containing a total of 45519 classes.

Palomba et al. [77] built a bug prediction model using code smell information as well. They combined the severity of the code smells involved (God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling, Message Chains) with existing bug prediction models. They found that the accuracy of the models increases by adding the code smell information as a predictor.

### 12.2.2 PCA and Correlation

Principal Component Analysis (PCA) is a widely used data reduction technique [89]. It reduces the dimensions of a large data set – in which the variables are interrelated – while retaining the variance of the data as much as possible. An orthogonal transformation is used to calculate the so called principal components, which are a set of linearly uncorrelated variables calculated from the possibly correlated variables of the data. The first principal component holds the greatest possible variance, each subsequent component will have the greatest variance remaining. PCA can be used for dimensionality reduction in large datasets, and it can uncover patterns in the data as well [17, 27].

In this research, we used PCA to examine the variance represented by the new metrics. We also calculated the correlation of the metrics to study the magnitude and direction of their linear relationships [47].

## 12.3 Evaluation

We formulated three research questions to examine the effect on bug prediction and the dimensionality of the PO metrics. In this section, we enumerate and answer these questions.

### 12.3.1 RQ1: How does adding the PO metrics affect the prediction capability of the original dataset?

Our first attempt was to merge the PO metrics into the original dataset. We calculated the previously introduced class level PO metrics for the 58 Java systems. The measurements were performed by extending the OpenStaticAnalyser (OSA) [5], which is an open source static analyser tool developed by the University of Szeged. Only those classes were kept in the dataset that were present in the original dataset as well, i.e. the classes that have bug information. We performed a 10-fold cross validation training with Weka [40] using the J48 algorithm on the original and on the extended dataset. Although many classifiers were applied for bug prediction [55], [59], Ferenc [37] used the J48 algorithm for the evaluation of the original dataset, therefore, we decided to utilize the same algorithm with its default parameters as well.

The results of the two datasets are compared in Table 12.1. The first column shows the values of the original dataset, while the second column shows the values of the extended dataset. In the rows, we see the most important attributes that characterize the performance of the J48 classifier: the precision[1], the recall[2], the F-measure[3], and the AUC[4]. It can be seen that the numbers show stagnation or a slight decrease. These changes are not significant. It causes changes of similar magnitude if the random seed is changed in the 10 fold cross validation.

We concluded that adding the PO metrics to the original dataset does not improve the overall prediction ability. Seeing this result, we decided to examine whether there is an improvement in cross project validation. We analyze this in the next RQ.

### 12.3.2 RQ2: Does including PO metrics increase the cross project bug prediction capability of the metric set?

With this RQ, we wanted to examine if there is a project on which a better model could be built using PO metrics. A project-wise evaluation was performed, meaning

[1]The precision can be calculated as the number of true positive elements divided by the sum of the true and false-positive elements. It describes what portion of the identifications was actually correct.

[2]It describes what portion of the actual relevant elements was identified correctly.

[3]F-measure is the harmonic mean of precision and recall.

[4]The ROC curve displays the performance of a classification model at all classification thresholds. The Area under the ROC curve (AUC) means the area under this curve. It is an aggregated measure of the performance across every classification thresholds.

| | Original | Extended |
|---|---|---|
| Precision for the not bugged class | 0.876 | 0.876 |
| Precision for the bugged class | 0.537 | 0.537 |
| Recall for the not bugged class | 0.919 | 0.919 |
| Recall for the bugged class | 0.421 | 0.419 |
| F-measure of the not bugged class | 0.897 | 0.897 |
| F-measure of the bugged class | 0.472 | 0.470 |

**Table 12.1:** The comparison of the results of the original and extended data set for J48 classifier

that we trained a model on each project and evaluated it on every other project. Out of the projects that were included in the database more than once, we used the ones with the highest version number. We found that the results for the different versions of the same systems are quite similar, therefore, they would have too much weight in the cross project validation. A total of 29 projects were used, so each model was evaluated on 28 systems. Each training was performed on the original dataset and on the extended dataset as well and the differences in the results were examined.

The F-measure difference between the evaluations performed on the original and the expanded dataset is shown in Table 12.2. Here, we used the Weighted Average F-measure[5] to compactly present the change of the F-measure values for both classes. The examination was performed with the J48 algorithm as well. The first column on the left shows which system the teaching was done on, and the column labels indicate which system the trained model was evaluated on. The second column contains the ID we gave to the system. These IDs are used in the column headers. The third column, called Class, shows how many classes are in the system, while the fourth column indicates the percentage of bugs. A colourmap is applied to these columns: in the third column, large class numbers are highlighted in orange, while in the fourth column orange colour represents a low bug ratio and blue colour represents a high bug ratio. We sorted the systems in such a way that the bug rate increases from top to bottom and from left to right. The green-coloured cells indicate a significant improvement, while values written in green denote slight improvements (less than 0.05). Due to the subsequent rounding, the value of 0.05 may appear in the table with both notations. For cells with a red background and values written in red, the same rule applies but in the negative direction. They indicate a slight or a significant decrease (larger than 0.05 in absolute value) in the F-measures of the expanded dataset.

It can be seen that if there were a disproportionate number of non-bugged classes (MCT, Neo4J) or bugged classes (Log4j 1.2, Xalan 2.7) in a system, then the models will be unusable on other systems. This is the explanation for the empty cells. The models are too fitted to one of the classes, therefore, no items are classified to the other class and the F-measure cannot be calculated. Where there is a 0.00 in the cell, the change in the F-measure is so insignificant that it does not appear in the rounded value. Significant F-measure reductions can be seen in the right half of the table. This can be explained by the fact that the models were built on systems with a low bug

[5]Weka calculates the Weighted Avg. F-Measure with the following formula for an $n$ and $y$ class: $Weighted\ Avg.\ F-measure = \frac{F-measure(n)\cdot NumOfInstances(x)+F-measure(y)\cdot NumOfInstances(y)}{NumOfInstances(n)+NumOfInstances(y)}$, where $NumOfInstances(n), NumOfInstances(y)$ correspond to the number of instances in the given class.

count, while in the last few columns the bug count of the systems is high, therefore, the models cannot function well.

Most of the significant improvements were achieved with the Xerces 2.0 model. It has a lower bug ratio then the Xalan 2.7 system, but still has plently of examples to learn and has 1.8 times more classes for the J48 algorithm. The model of the mcMMO project has also been significantly improved, at least for projects with similarly low bug ratios. Out of the 841 cells containing F-measure changes, 350 cells hold zero value and there are 126 blank ones (including the diagonal elements). The F-measure is reduced in 170 cases and improved in 195 cases. There are 45 significant F-measure increases in contrast to the 24 significant F-measure decreases.

Overall, the addition of PO metrics brings more improvement than deterioration in case of cross project validation, but it is important to choose the right system to build the model on. Further research is needed for the Xerces 2.0 and the mcMMO systems.

### 12.3.3 RQ3: Do the three PO metrics add valuable information to the dataset?

We calculated the correlation matrix[6] of the metrics and used Principal Component Analysis (PCA) to show the dimensionality of the PO metrics.

Table 12.3 only shows those parts of the correlation matrix which are relevant for the PO metrics. For better readability the table is divided into two parts. We used a colourmap to help better understand the correlations between metrics. The greater the positive correlation between two metrics the greener the cell is. Values close to 1 are labeled with pure green colour. A negative correlation would be marked in red, but there are no values close to -1 in the tables. If the value is zero or close to zero (meaning that the correlation is insignificant) the cells have a white background. It can be seen in the tables that there is no significant correlation, neither within the PO metrics, nor between the PO metrics and the other metrics.

In the following paragraphs, the results of the PCA will be discussed. The bug information was omitted for the calculations, but the other attributes were retained. Figure 12.1 depicts the cumulative variability for the principal components. It can be seen that the curve is gradual, i.e., a high number of principal components are required to maintain much of the variance. To cover 99% of the variance in the dataset, 33 principal components are required, which is more than half of the original attributes. Even 95% requires 20 factors.

To further investigate the significance of PO metrics we computed the so called factor loadings. A loading represent how much each original attribute contributes to the corresponding principal component. Table 12.4 shows the loadings for the first 20 principal components. Cells with an absolute value of 0.7 or greater were coloured with dark green. The absolute value of the bright green fields is between 0.5 and 0.7. Since there were very few outstanding values, only the rows where they occurred are shown in the table. To illustrate how small the loading values in the table are, we took the absolute value of the table and calculated the average and the median values. The average is 0,086, while the median is 0,06. It is clear that the 20 factors are made up of many metrics, and only a few of them have a prominent role in the composition of a factor. The PO metrics contribute significantly to components 13, 18, 19. That is,

[6]Correlation shows the degree to which a pair of variables is linearly related. We used the Pearson correlation for our experiment.

| Name | ID | Class | Bug | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| mct | 1 | 1 887 | **0.5** |  | 0.00 | 0.00 | 0.00 |  |  |  | 0.00 | 0.00 | 0.00 | 0.00 |  | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |  |  | 0.00 | 0.00 |  | 0.00 | 0.00 |
| neo4j | 2 | 5 899 | **1.0** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| jedit4.3 | 3 | 439 | **2.5** | 0.00 | 0.00 |  | 0.00 |  | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |  | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |  | 0.00 |  |  |  |  |  |  |  |  |  |
| ceylon | 4 | 1 610 | **4.2** | 0.00 | 0.00 | 0.00 |  | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| antrl | 5 | 479 | **4.4** | -0.01 | 0.00 | 0.01 | 0.00 |  | 0.00 | -0.01 | 0.00 | 0.00 | 0.01 | 0.02 | 0.00 | 0.01 | 0.00 | 0.00 | 0.00 | 0.00 | -0.01 | -0.01 | 0.00 | 0.01 | -0.07 | 0.05 | 0.00 | -0.05 | 0.07 | -0.04 | 0.01 | -0.11 |
| junit | 6 | 731 | **4.8** | 0.01 | 0.00 | -0.03 | 0.00 | -0.01 |  | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.02 | 0.01 | 0.00 | -0.01 | -0.02 | 0.09 | 0.00 | 0.00 | 0.05 | 0.04 | -0.05 | -0.06 | 0.02 | -0.09 | -0.09 | -0.04 | -0.24 |
| titan | 7 | 1 468 | **6.5** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |  | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| hazelcast | 8 | 3 412 | **11.0** | 0.00 | 0.01 | 0.00 | -0.02 | 0.01 | -0.01 | 0.00 |  | 0.00 | -0.02 | 0.00 | 0.01 | 0.01 | -0.01 | 0.00 | 0.01 | -0.03 | -0.03 | 0.00 | -0.02 | -0.02 | 0.04 | -0.01 | 0.01 | -0.02 | 0.02 | -0.04 | 0.09 | 0.18 |
| elasticsearch | 9 | 5 908 | **11.5** | 0.00 | 0.00 | -0.02 | 0.01 | 0.00 | 0.00 | 0.00 | 0.00 |  | 0.02 | 0.01 | 0.03 | 0.01 | 0.01 | 0.00 | -0.01 | -0.02 | 0.00 | 0.03 | -0.01 | 0.03 | -0.01 | 0.06 | 0.01 | -0.03 | -0.01 | -0.02 | -0.02 | 0.01 |
| mapdb | 10 | 331 | **12.1** | 0.00 | 0.01 | 0.00 | 0.00 | 0.01 | 0.00 | 0.00 | 0.01 | 0.01 |  | 0.01 | 0.00 | 0.01 | 0.01 | 0.01 | 0.00 | -0.01 | 0.02 | 0.01 | 0.01 | 0.00 | -0.01 | -0.02 | -0.01 | -0.02 | -0.11 | -0.07 | 0.01 | 0.01 |
| ivy2.0 | 11 | 294 | **12.6** | 0.00 | 0.00 | -0.04 | 0.00 | 0.00 |  | 0.00 | 0.00 | 0.00 | -0.01 |  | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.02 | 0.00 | 0.02 | 0.01 | 0.01 | -0.01 | 0.01 | 0.01 | 0.05 | 0.02 | 0.04 | 0.04 | 0.00 |
| oryx | 12 | 533 | **13.9** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |  | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| pdeui | 13 | 1 491 | **14.0** | -0.01 | 0.00 | -0.01 | 0.00 | -0.01 | 0.00 | 0.00 | 0.00 | -0.01 | -0.01 | 0.00 | -0.01 |  | -0.01 | 0.00 | 0.02 | -0.02 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.03 | 0.02 | 0.02 | -0.01 | 0.03 |
| myly | 14 | 1 405 | **14.9** | 0.00 | -0.01 | 0.00 | -0.01 | -0.01 | -0.01 | 0.00 | 0.00 | -0.01 | -0.02 | -0.03 | 0.00 | -0.01 |  | -0.01 | 0.00 | 0.01 | -0.02 | -0.01 | -0.01 | -0.04 | 0.00 | 0.00 | 0.01 | -0.02 | 0.00 | -0.05 | 0.04 | -0.02 |
| orientdb | 15 | 1 847 | **15.2** | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.00 | -0.01 | 0.00 |  | 0.00 | -0.01 | -0.01 | 0.00 | 0.00 | -0.03 | 0.04 | -0.02 | 0.00 | -0.02 | 0.00 | 0.01 | 0.00 | -0.03 |
| broadleaf | 16 | 1 593 | 18,3 | 0.00 | 0.00 | 0.00 | 0.00 | -0.01 | 0.00 | 0.00 | 0.01 | 0.00 | -0.02 | 0.01 | -0.01 | 0.01 | 0.00 | -0.01 |  | 0.02 | -0.01 | 0.01 | -0.01 | 0.04 | 0.00 | 0.00 | 0.00 | -0.04 | -0.05 | 0.03 | -0.02 | -0.07 |
| mcMMO | 17 | 301 | 18,9 | 0.03 | 0.03 | 0.06 | 0.05 | 0.04 | 0.01 | 0.04 | 0.05 | 0.05 | 0.05 | 0.02 | 0.01 | 0.07 | 0.04 | 0.05 | 0.02 |  | 0.06 | 0.03 | 0.02 | 0.09 | 0.02 | -0.04 | 0.00 | -0.07 | -0.01 | -0.13 | -0.11 | -0.18 |
| Eclipse JDT | 18 | 997 | 20,7 | -0.01 | 0.01 | -0.01 | 0.01 | 0.03 | 0.01 | 0.01 | 0.00 | 0.00 | 0.00 | 0.01 | 0.00 | -0.01 | -0.01 | -0.02 | -0.01 | 0.06 |  | 0.00 | 0.06 | -0.01 | 0.01 | 0.03 | -0.01 | -0.01 | 0.03 | 0.02 | 0.02 | 0.03 |
| camel1.6 | 19 | 795 | 21,4 | 0.00 | 0.00 | -0.03 | 0.00 | 0.02 | -0.01 | 0.01 | 0.00 | 0.01 | 0.01 | -0.02 | 0.00 | 0.01 | 0.00 | 0.00 | -0.01 | -0.03 | -0.03 |  | 0.00 | -0.04 | -0.06 | -0.09 | 0.02 | 0.01 | 0.08 | 0.04 | 0.06 | 0.14 |
| netty | 20 | 1 143 | 23,7 | 0.00 | 0.00 | 0.00 | 0.00 | -0.01 | 0.00 | 0.00 | -0.01 | -0.01 | 0.00 | 0.00 | -0.02 | 0.00 | -0.01 | 0.01 | 0.00 | 0.00 | 0.00 | -0.01 |  | 0.02 | 0.06 | 0.00 | -0.01 | -0.01 | -0.02 | -0.01 | -0.02 | 0.01 |
| ant1.7 | 21 | 681 | 24,2 | 0.01 | 0.00 | 0.05 | 0.01 | 0.01 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | -0.02 | -0.01 | -0.01 | 0.00 | 0.01 | 0.00 | -0.02 | -0.01 | 0.00 | 0.00 |  | 0.00 | -0.02 | -0.04 | -0.02 | -0.02 | -0.02 | 0.00 | 0.00 |
| velocity1.6 | 22 | 188 | 35,1 | 0.00 | 0.00 | 0.01 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | -0.03 | 0.00 | 0.01 | 0.01 | 0.00 | -0.04 | -0.03 | 0.01 | 0.01 |  | 0.01 | 0.04 | 0.02 | 0.00 | 0.03 | 0.02 | 0.08 |
| synapse1.2 | 23 | 228 | 36,8 | -0.01 | -0.01 | -0.03 | -0.02 | -0.01 | 0.00 | -0.02 | -0.01 | 0.00 | -0.05 | -0.01 | -0.01 | 0.00 | -0.01 | -0.02 | 0.00 | 0.00 | -0.05 | -0.01 | -0.03 | -0.01 | -0.01 |  | -0.05 | 0.02 | 0.04 | -0.07 | 0.00 | -0.02 |
| equinox | 24 | 319 | 39,5 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |  | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| lucene2.4 | 25 | 320 | 60,3 | 0.02 | 0.00 | 0.02 | 0.02 | -0.01 | 0.00 | -0.01 | 0.00 | -0.02 | 0.01 | -0.03 | 0.00 | -0.04 | 0.00 | -0.01 | 0.01 | 0.00 | -0.13 | -0.02 | 0.02 | 0.00 | 0.00 | 0.01 | 0.07 |  | 0.00 | 0.04 | 0.00 | 0.02 |
| poi3.0 | 26 | 414 | 66,7 | 0.01 | 0.01 | 0.00 | 0.00 | 0.01 | 0.02 | 0.00 | 0.00 | 0.00 | 0.02 | 0.02 | 0.00 | 0.00 | 0.01 | 0.01 | 0.01 | 0.02 | -0.01 | 0.01 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |  | 0.01 | 0.00 | 0.00 |
| xerces2.0 | 27 | 342 | 88,6 | 0.25 | 0.28 | 0.21 | 0.34 | 0.09 | 0.26 | 0.39 | 0.22 | 0.16 | 0.24 | 0.18 | 0.40 | 0.24 | 0.28 | 0.16 | 0.12 | 0.28 | 0.12 | 0.09 | 0.17 | 0.12 | 0.14 | 0.07 | 0.09 | 0.02 | -0.16 |  | -0.10 | -0.01 |
| log4j1.2 | 28 | 191 | 91,6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| xalan2.7 | 29 | 848 | 98,7 |  | 0.00 |  |  |  |  | 0.00 | 0.00 | 0.00 |  |  |  |  |  |  |  |  |  |  | 0.00 |  |  |  |  | 0.00 |  | 0.00 |  |  |

**Table 12.2:** Change in Weighted Average F-measure values for the cross project validation

| | MPC | SFP-SCU | SFP |
|---|---|---|---|
| MPC | 1.00 | 0.09 | 0.29 |
| SFP-SCU | 0.09 | 1.00 | 0.23 |
| SFP | 0.29 | 0.23 | 1.00 |
| CLOC | 0.40 | 0.17 | 0.45 |
| NOD | 0.02 | 0.03 | 0.01 |
| TNPA | 0.04 | 0.04 | 0.17 |
| NLPM | 0.43 | 0.04 | 0.27 |
| TNLPA | 0.02 | 0.05 | 0.25 |
| NLPA | 0.01 | 0.05 | 0.24 |
| TNOS | 0.41 | 0.14 | 0.44 |
| TLLOC | 0.47 | 0.15 | 0.51 |
| LLOC | 0.48 | 0.15 | 0.52 |
| CLC | 0.01 | 0.00 | 0.03 |
| WMC | 0.44 | 0.14 | 0.42 |
| CCL | 0.17 | 0.06 | 0.24 |
| CC | 0.02 | 0.00 | 0.04 |
| TNA | 0.08 | 0.07 | 0.21 |
| NOC | 0.02 | 0.01 | 0.00 |
| PDA | 0.38 | 0.05 | 0.24 |
| TLOC | 0.48 | 0.17 | 0.55 |
| CCO | 0.13 | 0.07 | 0.19 |
| TNS | 0.10 | 0.02 | 0.10 |
| NA | 0.07 | 0.07 | 0.20 |
| NLA | 0.10 | 0.11 | 0.34 |
| DLOC | 0.40 | 0.15 | 0.31 |
| NPA | 0.03 | 0.04 | 0.16 |
| NII | 0.23 | 0.10 | 0.11 |
| NLS | 0.18 | 0.03 | 0.11 |
| LLDC | 0.22 | 0.04 | 0.27 |
| TNLS | 0.18 | 0.03 | 0.12 |
| CI | 0.15 | 0.05 | 0.21 |
| TCLOC | 0.40 | 0.17 | 0.45 |
| NLM | 0.48 | 0.10 | 0.35 |
| TNPM | 0.18 | 0.01 | 0.16 |
| NG | 0.14 | 0.01 | 0.13 |
| NLG | 0.32 | 0.05 | 0.23 |
| TNLA | 0.11 | 0.11 | 0.35 |
| NOA | 0.02 | -0.01 | 0.02 |
| TNM | 0.18 | 0.06 | 0.19 |
| TCD | 0.07 | 0.03 | 0.07 |
| AD | 0.10 | 0.04 | 0.11 |
| PUA | 0.23 | 0.00 | 0.14 |
| LDC | 0.22 | 0.04 | 0.26 |
| NL | 0.23 | 0.09 | 0.27 |
| CBO | 0.25 | 0.07 | 0.31 |
| NS | 0.09 | 0.01 | 0.07 |
| NOP | 0.03 | -0.01 | 0.02 |
| CLLC | 0.02 | 0.00 | 0.04 |
| LOC | 0.49 | 0.17 | 0.55 |
| DIT | -0.01 | -0.01 | 0.01 |
| NPM | 0.19 | 0.01 | 0.15 |
| TNLM | 0.45 | 0.11 | 0.35 |
| NOI | 0.33 | 0.07 | 0.35 |
| LCOM5 | 0.18 | 0.01 | 0.07 |
| TNLPM | 0.41 | 0.04 | 0.28 |
| NM | 0.21 | 0.05 | 0.19 |
| CD | 0.07 | 0.03 | 0.07 |
| NOS | 0.42 | 0.14 | 0.44 |
| TNLG | 0.30 | 0.06 | 0.23 |
| CBOI | 0.17 | 0.06 | 0.09 |
| RFC | 0.45 | 0.10 | 0.40 |
| TNG | 0.13 | 0.03 | 0.14 |
| NLE | 0.24 | 0.08 | 0.26 |
| bugs | 0.12 | 0.04 | 0.18 |

**Table 12.3:** Correlation between the metrics, only containing the columns that are relevant for the PO metrics

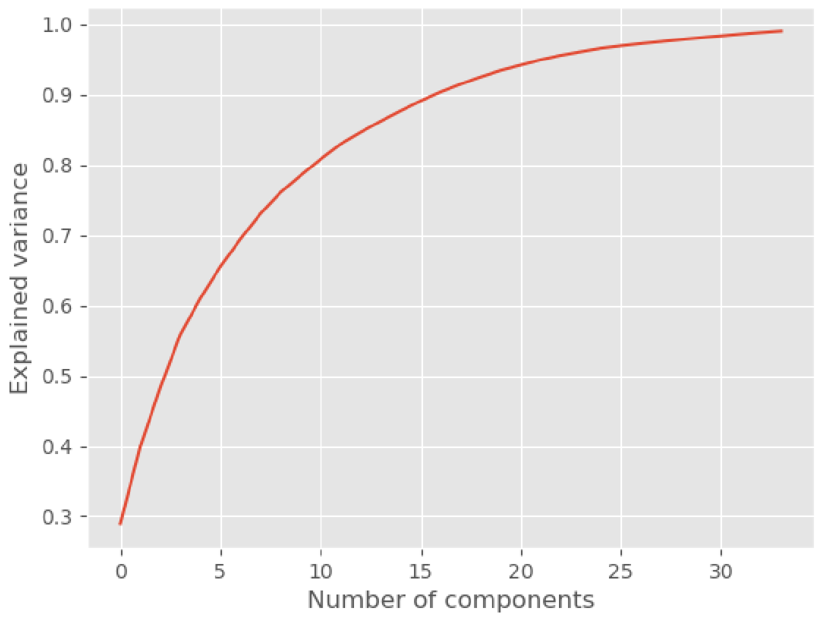

**Figure 12.1:** Variability of the principal components

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MPC | 0.12 | 0.04 | -0.04 | -0.07 | 0.03 | -0.02 | -0.07 | -0.06 | 0 | 0.16 | 0.04 | 0.12 | 0.03 | 0 | -0.09 | -0.09 | 0.53 | -0.13 | 0.73 | -0.11 |
| SFP-SCU | 0.03 | 0.03 | 0.03 | -0.03 | -0.02 | -0.01 | -0.02 | -0.03 | 0.08 | 0.01 | 0.12 | 0.29 | 0.06 | 0.82 | 0.1 | -0.18 | -0.21 | 0.11 | 0.02 | -0.31 |
| SFP | 0.12 | 0.07 | 0.08 | -0.03 | 0 | -0.01 | 0 | -0.11 | 0.14 | 0.04 | 0.08 | 0.16 | 0.09 | 0.24 | 0.2 | 0.06 | 0.14 | -0.26 | -0.09 | 0.79 |
| NII | 0.08 | -0.01 | -0.03 | -0.14 | 0.07 | 0 | -0.21 | 0.33 | 0.08 | 0 | 0.03 | 0.05 | -0.01 | 0.13 | -0.55 | 0.05 | -0.03 | 0.02 | -0.03 | 0.15 |
| TNLG | 0.14 | -0.09 | -0.08 | -0.11 | 0.14 | 0.09 | 0.07 | 0.01 | -0.07 | -0.02 | 0.07 | -0.03 | 0.51 | -0.09 | 0.06 | -0.06 | -0.17 | 0.03 | 0.08 | 0 |

**Table 12.4:** Factor loadings of the first 20 factors only showing the significant values

using PCA analysis, we concluded that PO metrics contributed notably to the variance of the dataset.

## 12.4 Future work

The results are based on only one learning algorithm. Further studies with other classifiers are needed for the full picture. It is possible that for different learning algorithms, such as Random Forest or KNN, the extended dataset would yield different results.

The bug prediction database we used is widely known and accepted, however, it is relatively old and does not have a system newer than Java 8. Although we discarded projects that did not have a sufficient number of classes, there were still some systems in which the bug ratio was too low or too high to build a usable model. For real applications, a bug rate above 90% is unlikely. Our results also confirmed the idea that these systems are not effective for bug prediction. Additional databases should be tested for comparison.

The calculation of the PO metrics is currently only implemented for one programming language, Java. In order to fully evaluate their usefulness, it is necessary to adapt them to other languages as well, e.g. C ++, JavaScript. However, in addition to the implementation tasks, building a bug database for these languages would also be a challenge.

## 12.5 Conclusion

In this research, three Primitive Obsession metrics were investigated in more detail: the Method Parameter Clones ($MPC$), the Static Final Primitives ($SFP$), and the Static Final Primitive - Switch Case Usage ($SFP - SCU$) metric. We examined the usefulness of these new metrics by integrating them into an existing bug prediction dataset and checking how they change the bug prediction ability of the dataset. RQ1 showed that adding them to the entire dataset did not result in an improvement, however, in case of cross project bug prediction they significantly contributed to the improvement of multiple prediction models. Even though they also caused a decrease in the predictive ability of some models, the number of improvements is larger than the number of declines by an extent of 25. That is, the PO metrics bring improvement in 28,5% of the cases, while the amount of deterioration is 23%. For the Xerces 2.0 system, the improvement was remarkably high on systems that have a low bug count. Adding PO metrics brings improvement in 90% of cases and a decrease in only 7% of cases.

With RQ3 we found that the PO metrics indeed contribute to the variance of the dataset. They did not correlate with the previous source code metrics. It can be seen that PO metrics are a new, useful approach for describing and measuring the source code.

# 13
# Summary of Part III

In the previous chapters, I presented and studied possible methods for the automatic detection of Primitive Obsession. This code smell represents the overuse of primitive data types. This may seem like a clear definition, but it is not sufficient for automatic detection. For this reason, in the initial stages of our research, we proposed various definitions to try to capture the characteristics of Primitive Obsession-infested source code.

I defined a metric called *Method Parameter Clones (MPC)* to detect the repetitive parameter type-name pairs of the methods of a class. This is useful because if related data is not encapsulated in a structure or a class, they tend to appear together in the parameter list of the methods that use them. With the *Static Final Primitive Usage (SFP)* metric, I aimed to detect a typical Primitive Obsession manifestation, type codes. Type codes are when the type-dependent behavior of a class is modeled by constants stored in primitive data types. The metrics count the usage of possible type code values for a class. A specialization of SFP is the *Static Final Primitives - Switch Case Usage (SFP-SCU)* metric, which specifically examines the occurrence of type codes in `switch-case` statements. I implemented these metrics along with additional PO metrics defined by my co-researcher in the OSA [5] static analyzer framework for Java. We analyzed three Java systems and an example project for Primitive Obsession. We manually validated the results. The results showed that the metrics I defined are suitable for detecting type codes and method signatures that indicate code that is overloaded with primitive types. Based on the manual inspection, we did not find any type codes in the systems that the SFP-SCU metric missed.

In a subsequent study, I further investigated the usability of the metrics with the help of an existing bug prediction database [37]. In addition to MPC and SFP-SCU, this research also includes the general SFP metric. I integrated these into the database, then compared the F-measure values of the J48 algorithm with the original, unmodified results. The numbers showed stagnation or a slight decrease. Although the PO metrics do not improve the overall prediction ability of the original dataset, the cross-project validation showed a different picture. We trained a model on each project and evaluated it on every other project. That is, a total of 29 trainings were done, and all models were evaluated on 28 systems. Although this did not yield an F-measure improvement for all systems, in the case of multiple systems, the extended models delivered significant improvements over the original models. To conclude, PO metrics can contribute to improving bug prediction, but the selection of the appropriate training systems requires further research. A correlation test and a PCA analysis between the metrics of the original dataset and the PO metrics were also performed. The correlation test showed no significant linear relation, and the PCA showed that the PO metrics contribute to the variance of the data set.

In this thesis point, we introduced several new Primitive Obsession metrics. We proved that they are a new, useful approach for describing and measuring the source code and they are capable of improving the bug prediction ability.

*"However insignificant we may be. We will fight! We will sacrifice! And we will find a way! That's what humans do!"*

— Commander Shepard *(Mass Effect)*

# 14 Conclusions

In this thesis, three main topics were discussed. All three topics are related to static source code analysis and quality assurance. The first is in connection with symbolic execution, the second topic is about Java static call graphs, and the third introduces Primitive Obsession detection techniques.

In the case of *symbolic execution*, we enhanced an existing Java symbolic execution engine with heuristical solutions. We targeted two important areas with these heuristics. One is related to the handling of static initialization blocks, the other is the introduction of a simplified constraint solver. We provided a lightweight but well-functioning solution to both problems, eliminating countless false-positive error messages. We did not increase the resource requirements of the engine, but we did increase its accuracy. By pruning unnecessary execution branches, we have made it possible to discover new, true positive warnings as well. The solutions have been tested on more than two hundred Java systems and became a persistent part of the symbolic executive engine.

In the second part of the thesis, we analyzed *static call graph* generator tools for Java. Our goal was to make an in-depth, qualitative and quantitative comparison using the most important open-source static analyzer tools that can also be used for call graph creation. As a first step in the comparison, we developed a heuristical call graph matching algorithm. This was necessary because the tools often indicated the same methods differently. For example, the names of the constructors and initialization blocks were not uniform, not to mention the methods of the anonymous classes and the lambda functions. Once the namings were standardized, it became possible to explore the actual differences between the graphs. Our studies revealed significant differences between the graphs. The discrepancies come from three main sources: the handling of Java language elements (e.g., initializer blocks, lambdas), the processing differences (such as the handling of library methods), and the algorithmic differences (like the handling of polymorphism). These are the main factors that can significantly determine the generated call graphs. This thorough comparison can help developers find the appropriate call-graph creator algorithm for their purpose.

Finally, the last part of the thesis deals with *Primitive Obsession.* We examined the characteristic of this code smell and based on them we defined several class-level metrics. The metrics cover properties such as repetitive method parameter lists and type code usage. We implemented the metrics in a Java static source code analyzer and tested them on a small sample code and on three real-sized systems. The reported classes and methods were examined manually. We then integrated the metrics into a bug prediction database and examined the changes on the J48 algorithm. In certain cases, the metrics were capable of improving the bug prediction ability. The PCA analysis showed that they contribute to the variance of the dataset. We concluded that the introduced class-level Primitive Obsession metrics are a new, useful approach for describing and measuring the source code.

## Future Work

Naturally, the results achieved in this thesis can be extended and further developed. Some of the improvements are already underway, while the rest are planned for the near future.

We intend to continue our research on RTEHunter in the coming period of time. In addition to adapting the engine to the latest Java version, we also want to revise the results for false-positive warnings. We want to examine whether it is possible to refine the engine with additional heuristical solutions. The information gained from researching call graphs can also help to push this research forward. The accuracy of symbolic execution depends on how method calls are resolved. The call graph comparison shed light on the factors that we want to examine in RTEHunter.

In the case of the call graph-related research, the comparison of static call graphs with dynamic call graphs is already in progress. With this comparison, our goal is to determine the extent to which the call graphs generated by static analyzers cover the dynamic call graphs generated by test suite execution. We want to answer whether dynamic call graph generation can be replaced by a combination of static tools and the inaccuracies of this replacement. Due to the limited code coverage of the tests, dynamic call graphs do not include all edges either. Therefore, it is important to consider what is contained only in graphs generated by static analysis.

Primitive Obsession metrics can also be improved. It would be interesting to examine how useful developers find the warnings provided by these metrics integrated into a development environment. In addition to examining the user experience, we also plan to carry out further studies based on the results of bug prediction research. We need to look more closely at what caused the large improvements or deteriorations. It may also be interesting to perform tests with a smaller set of metrics. This would mean that in addition to Primitive Obsession metrics, fewer conventional metrics would be used in the bug prediction. This would allow us to examine the extent to which Primitive Obsession metrics can replace a more difficult-to-calculate, traditional metric. Another interesting research would be to apply our Primitive Obsession metrics to a language other than Java.

## Epilogue

I have spent the last few years of my life researching source code quality assurance. This helped me to see my own development work from a different perspective. I pay attention to always producing maintainable code and I try to pass this on to my students as well.

Of the three topics, research on Primitive Obsession came closest to my heart. It was fantastic to explore an area not yet studied before us. Like when an adventurer enters an uncharted land. It was a wonderful feeling to discover new things and I want to continue doing so in the future as a researcher.

# Appendices

# A
# Summary in English

Static analysis is a crucial part of software development. Not only does it help to detect errors in an early stage, but it also supports maintaining proper code quality. The thesis focuses on three main topics that emphasize the importance of static analysis. In the first part, we present two heuristical methods in order to improve the Java symbolic execution engine called RTEHunter. The second part presents a more crucial domain, the in-depth analysis of static call graph creation tools, while the third part discusses the detection of the Primitive Obsession code smell. The resulting statements are grouped into three major thesis points. The relation between the underlying supporting publications and the thesis points is presented in Table A.1.

**I. Heuristical Improvements of a Symbolic Execution Engine**

The contributions of this thesis point – related to symbolic execution – are discussed in Chapters 2 - 5. We conducted research with a Java symbolic execution tool called RTEHunter.

*1. Improving Static Initialization Block Handling in Java Symbolic Execution Engine* The handling of the static initialization blocks in Java is hard to simulate in the synthetic environment of symbolic execution. We can think of these blocks as functions called automatically during the loading of the class. For a static analyzer tool like RTEHunter that does not actually execute the Java bytecode, it is difficult to handle the Java classloading mechanism. Because of this, static initialization blocks were completely neglected during execution leading to plenty of false-positive error messages, especially `NullPointerExceptions` (NPE). We came up with a filtering mechanism that fits well with RTEHunter's method-by-method-based implementation. When an NPE arose, we - instead of mimicking the behavior of the Java ClassLoader - simply checked if the variable associated with this warning was initialized in the static initializer blocks of its class. If the answer was positive the warning was treated as a false-positive. This approach was tested on 209 various-sized Java systems. Originally there were 3,369 NPE warnings on these systems. The manual verification showed that not only did we eliminate 242 erroneous NPE warnings, but also discovered 7 true positive runtime issues, which is about a 7% improvement.

*2. Applying Heuristics to Improve our Java Symbolic Execution Engine.* The basic idea of symbolic execution is that the program is executed on symbolic values instead of concrete ones. If a conditional statement is reached during execution both the true and false execution paths will be discovered. From these conditional statements, constraints can be derived to bind the values of symbolic variables. Theoretically, every possible execution path will be explored. However, based on the conditions associated with the symbolic variables, there will be paths that can be pruned as their conditions are infeasible. The logical expression that can be built from the constraints associated with symbolic variables is called the path condition. The path condition is a key part of symbolic execution, however, its maintenance and feasibility check can be extremely resource-intensive. They consume a big portion of the overall runtime causing symbolic engines to scale poorly on bigger systems. In this research, we applied a heuristical constraint handling mechanism called the null constraint solver. This solver tracks whether the value of a symbolic Java reference is null throughout its lifetime. This approach did not notably alter the computational time requirements of RTEHunter as no complicated feasibility check is needed. To see the improvement, we executed both the original, constraint solverless and the improved versions of RTEHunter on 209 Java systems. Our conclusion is that not only did we eliminate 16 serious false-positive warnings (out of 3,102), but also discovered 7 true positive runtime issues.

In this thesis point, my goal was to develop methods to increase the accuracy of the indicated errors without further increasing the resource requirements of the already computationally intensive process. Analyses performed on more than two hundred systems have confirmed that this aim was successfully achieved.

***The Author's Contributions***

The author worked on the theoretical development of the two heuristic solutions: the handling of the static initialization blocks and the null constraint solver. She performed the literature review in the field of symbolic execution to review the currently available technologies and methodologies. Following the conceptual design, the author implemented the heuristics in the RTEHunter system. She performed the necessary tests on smaller sample codes and then gathered more than 200 Java systems to analyze the impact of the enhancements. She has performed the analyses on the collected projects and completed the manual validation in both cases. The publications related to this thesis point are:

- **Edit Pengő** and István Siket. Improving Static Initialization Block Handling in Java Symbolic Execution Engine. In Computational Science and Its Applications – ICCSA 2017: 17th International Conference, Proceedings, Part V, pages 561-574. Springer, 2017.
- **Edit Pengő**. Applying Heuristics to Improve our Java Symbolic Execution Engine – ICAI 2017. In Proceedings of the 10th International Conference on Applied Informatics – ICAI 2017, pages 245-253. Eszterházi Károly Catholic University, 2017.

**II. Comparison of Static Call Graph Builder Tools**

The second thesis point - related to the examination of Java static call graphs - is discussed in Chapters 6 - 9. We selected six open-source static analyzers (SOOT [5], Spoon [78], OSA [5], WALA [6], JCG [43], and Soot [85]) to compare their call graphs.

*1. A Preparation Guide for Java Call Graph Comparison. Finding a Match for Your Methods.*

The way to compare the capabilities of call graph builder tools is by comparing their generated call graphs. Methods developed for general-purpose graphs cannot be directly applied to call graphs, especially if they were produced by different analyzer tools. Even if the structure of two call graphs is isomorphic, they can be considered completely different because of the labeling of the nodes. Therefore, we had to make the call graphs comparable by finding a mapping between their nodes. A heuristical algorithm was developed for this in multiple steps. The basis of the pairing is the methods' fully qualified name. It was refined by the so-called *anonymous transformation*. Anonymous source code elements have a non-standardized name, meaning that static analyzers can name the same code element differently. Anonymous transformation simply means that we replace the non-standardized part of their name with a constant string, and then a name-based pairing is performed. If a class has multiple anonymous classes, this approach causes a loss in the accuracy of the pairing. Our next step was a *line information-based* pairing. In addition to the method names, we also used their position in the source code for pairing. Unfortunately, line numbering is not as consistent among static analyzers as it could be expected, therefore, the usage of line information was restricted only for anonymous and generic source code elements, whilst, for traditional methods, the name-wise pairing was used. Lastly, we introduced a method to *handle Java's generic elements*. The problem is that if a generic method is instantiated with different types, tool A might represent it with multiple nodes, while tool B may represent it with only one node. We solved this problem by collecting all possible pairings. Combining the line information usage and the generic element handling the pairing has improved by 2-3 percent for some tools. This is a considerable enhancement, however, a significant number of nodes remained unmatched. We manually investigated the root causes of this. Our in-depth examination revealed that most of the unmatchings cannot be resolved. These are due to the differences in the operation of the tools.

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

After developing the pairing mechanism, we performed an in-depth analysis of the remaining differences. With this comparison, our aim was to help to take the appropriate considerations into account when developing a call graph-based algorithm. The reliability of the call graph can influence the results of subsequent processes. We challenged the six tools on an example code containing the language features of Java 8 and on multiple real-life open-source Java systems and performed a quantitative and qualitative assessment of the resulting graphs. We identified six sources for the differences: the handling of initializer methods, polymorphism, Java 8 language elements (such as lambdas), dynamic method calls, generic source code elements, and anonymous source code elements. The different treatment of these features can cause significant differences in the generated graphs.

Although the pairing algorithm does not pair all possible nodes because of the lack of line information, the differences are not primarily due to that. Knowing this, we removed those parts from the graphs that are due to known differences. We examined the results of multiple filtering attempts. Finally, we decided to keep only those methods in the graphs that were found by every tool. All other methods and their associated edges were filtered, but still significant differences remained between the graphs. Their manual examination revealed previously undiscovered factors such as errors or features of bytecode-based parsers. We proved that the outputs of the call graph builder tools would not be the same even by eliminating the known differences.

In this research, we implemented a call graph pairing algorithm. With its help, we revealed the extent and the causes of the differences between the call graphs. This comparative study can help developers find the appropriate call-graph creator algorithm for their purpose.

***The Author's Contributions***

The author participated in the selection of the call graph creator tools. She found a total of four that met the selection criteria: OSA, WALA, SPOON, and JDT. She tested these tools and implemented call graph exporters for them. The author worked on the theoretical development of the call graph comparison algorithm. She provided the ideas for the anonymous transformation, the line information-based pairing, and the handling of generic elements. She performed the literature review in the field of graph comparison to review the currently available technologies and methodologies. The author participated in analyzing and validating the results of the implemented call graph comparison tool. She also took part in the comparative study that followed. She updated the four call graph creator tools, if it was necessary and analyzed the selected Java systems with them. She helped to extend the sample code that included the Java 8 features. She also aided in interpreting the results and manually examining the differences in the graphs. She participated in forming the research questions and in the theoretical development of the steps for filtering the known differences of the graphs. The publications related to this thesis points are:

- **Edit Pengő**, Zoltán Ságodi and Ervin Kóbor. Who Are You not gonna Call? A Definitive Comparison of Java Static Call Graph Creator Tools. In The 11th Conference of PhD Students in Computer Science: Volume of short papers – CSCS 2018, pages 68-71. University of Szeged, 2018.
- Zoltán Ságodi and **Edit Pengő**. A Preparation Guide for Java Call Graph Comparison. Finding a Match for Your Methods. Published in Acta Cybernetica, Volume 24, No 1, Pages 131-155. 2019.
- Judit Jász, István Siket, **Edit Pengő**, Zoltán Ságodi and Rudolf Ferenc. Systematic Comparison of Six Open-source Java Call Graph Construction Tools. In Proceedings of the 14th International Conference on Software Technologies – ICSOFT 2019, pages 117-128. SciTePress, 2019.

## III. Development and Evaluation of Primitive Obsession Metrics

The third thesis point deals with the hitherto less studied code smell, Primitive Obsession. It is discussed in Chapters 10 - 13.

*1. Grasping Primitive Enthusiasm - Approaching Primitive Obsession in Steps.* In our preliminary study, we defined a metric called Primitive Enthusiasm (PE) that helps detect the overuse of primitive types in method parameters. In this work, we introduced additional metrics to describe more aspects of Primitive Obsession. Type code usage is detected by *Static Final Primitives (SFP)* and *Static Final Primitives - Switch Case Usage (SFP-SCU)*. The *Method Parameter Clones (MPC)* metric highlights those method parameters that could be extracted to a value object. The original PE metric was also further refined, with the introduction of two other metrics: *Global Primitive Enthusiasm (GPE)* and *Hot Primitive Enthusiasm (HPE)*. I examined how the results of the MPC and SFP-SCU metrics relate to the results of the PE variants. The metrics were implemented in the OpenStaticAnalyzer (OSA) [5] toolchain for Java and evaluated on three real-sized Java projects. We experimented with two method exclusion strategies to get more precise results. This investigation resulted in skipping every method with just one parameter. The reported warnings were sampled and manually investigated. The *MPC* metric captured repetitive method parameter lists thus Primitive Obsession-infected classes. As the PE variants report a high number of methods and classes, MPC can be used to weight these results. The combination of the PE and *MPC* metrics can be useful to prevent the appearance and spread of Primitive Obsession. The *SFP-SCU* metric detected multiple type code usages in the projects. The findings showed that the new metrics can highlight many smelly and hardly readable code segments.

*2. Examining the Bug Prediction Capabilities of Primitive Obsession Metrics.*

We integrated the three class-level Primitive Obsession metrics (the *MPC*, the *SFP* , and the *SFP-SCU* metrics) into an existing bug prediction database [37] and examined the effects. First, we evaluated the original and the extended bug datasets with the J48 algorithm and studied the F-measure[1] changes. The numbers showed stagnation or a slight decrease. We concluded that adding the PO metrics to the original dataset does not improve the overall prediction ability. The cross-project validation showed a different picture. We trained a model on each project and evaluated it on every other project. A total of 29 projects were used, so each model was evaluated on 28 systems. Each training was performed on the original dataset and on the extended dataset as well, and the differences in the results were examined. Out of the 841 cases, the F-measure is reduced in 194 cases and improved in 240 cases. Overall, the addition of PO metrics brings more improvement than deterioration in the case of cross-project validation, but it is important to choose the right system to build the model on. We also calculated the Pearson correlation matrix of the metrics and used Principal Component Analysis (PCA) [89] to show the dimensionality of the PO metrics. The correlation showed no significant linear relation neither within the PO metrics, nor between the PO metrics and the other metrics. The PCA calculation revealed that to cover 99% of the variance in the dataset, 33 principal components are required, which is more than half of the original attributes. Even 95% requires 20

[1]F-measure is the harmonic mean of precision and recall.

factors. We calculated the factor loadings [2] for these 20 components as well. The results showed that the PO metrics contribute significantly to three of them.

In this thesis point, we introduced several new Primitive Obsession metrics. We proved that they are a new, useful approach for describing and measuring the source code and they are capable of improving the bug prediction ability.

***The Author's Contributions***

The author participated in the selection of the analyzed systems and the evaluation of the original Primitive Enthusiasm metric. She performed the literature review in the field of code smells and Primitive Obsession. She designed the *MPC*, the *SFP*, and the *SFP-SCU* metrics. She re-implemented the Primitive Enthusiasm calculation and implemented the other five metrics' calculations for the extended research coming after the short paper. She experimented with method elimination techniques and decided to settle with skipping every method with just one parameter or less. She also participated in the selection of the Java systems and the assessment of the results. She examined the overlaps in the results of the metrics and investigated their correspondence. Later, the author implemented the Weka-based application for the bug dataset-related research. She executed the OSA static analyzer on 58 systems and integrated the results into the dataset. An evaluation program was created by the author to compare the results of the original and the expanded data set. It was entirely her job to formulate the RQs and to evaluate and interpret the results. The publications related to this thesis point are:

- Péter Gál and **Edit Pengő**. Primitive Enthusiasm: A Road to Primitive Obsession. In The 11th Conference of PhD Students in Computer Science: Volume of short papers – CSCS 2018, pages 134-137. University of Szeged, 2018.
- **Edit Pengő** and Péter Gál. Grasping Primitive Enthusiasm - Approaching Primitive Obsession in Steps. In Proceedings of the 13th International Conference on Software Technologies – ICSOFT 2018, pages 389-396. SciTePress, 2018.
- **Edit Pengő**. Examining the Bug Prediction Capabilities of Primitive Obsession Metrics. In Computational Science and Its Applications – ICCSA 2021: 21st International Conference, Proceedings, Part VII, pages 185-200. Springer, 2021.

Table A.1 summarizes the main publications and how they relate to the thesis points.

| № | [118] | [119] | [116] | [115] | [113] | [112] | [114] | [117] |
|---|---|---|---|---|---|---|---|---|
| I. | ♦ | ♦ | | | | | | |
| II. | | | ♦ | ♦ | ♦ | | | |
| III. | | | | | | ♦ | ♦ | ♦ |

**Table A.1:** Thesis contributions and supporting publications

[2]A loading represents how much each original attribute contributes to the corresponding principal component.

# B
# Magyar nyelvű összefoglaló

A statikus forráskód elemzés döntő szerepet játszik a szoftverfejlesztésben. Nemcsak a hibák korai felismerését segíti, hanem a megfelelő kódminőség fenntartását is támogatja. A disszertáció három fő témára összpontosít, amelyek közös pontja, hogy mindegyik hangsúlyozza a statikus elemzés fontosságát. Az első részben két heurisztikus módszert mutat be az RTEHunter nevű Java szimbolikus végrehajtó motoron. A második rész a statikus hívási gráf-létrehozó eszközök mélyreható összehasonlításával foglalkozik, míg a harmadik rész a Primitive Obsession gyanús kód (code smell) felismerését tárgyalja. Az elért eredményeket három fő tézispontba soroltam. A publikációk és a tézispontok közötti kapcsolatot a A.1. táblázat szemlélteti.

## I. Egy szimbolikus végrehajtó motor működésének javítása heurisztikus módszerekkel

Ez, a szimbolikus végrehajtással kapcsolatos tézispont a 2 - 5. fejezetekben van tárgyalva. Az RTEHunter nevű, Java szimbolikus végrehajtó eszközzel végeztünk kutatásokat.

*1. A statikus inicializáló blokkok kezelésének javítása egy Java szimbolikus végrehajtó motorban*

A Java statikus inicializáló blokkjainak kezelését nehéz szimulálni a szimbolikus végrehajtás szintetikus környezetében. Tekinthetünk ezekre a blokkokra úgy, mint olyan metódusokra, amelyeket automatikusan hívunk az osztály betöltése során. Egy olyan statikus elemző eszközben, mint az RTEHunter, amely valójában nem hajtja végre a Java byte-kódot, nehéz pontosan kezelni a Java osztálybetöltési mechanizmust. Emiatt korábban nem is kezelte a statikus inicializáló blokkokat, ami rengeteg fals pozitív hibaüzenetet eredményezett, különösen `NullPointerExceptions` (NPE) hibaüzeneteket. Olyan filterezési mechanizmust dolgoztunk ki, amely jól illeszkedik az RTEHunter metódusonként történő szimbolikus végrehajtásához. Ahelyett, hogy utánoznánk a Java ClassLoader viselkedését, egyszerűen ellenőrizzük egy NPE megjelenésekor, hogy a hozzá társított változó inicializálva lett-e osztályához tartozó statikus inicializáló blokkok egyikében. Amennyiben igen, az NPE-t fals pozitívként kezeljük. Ezt a megoldást 209 különböző méretű Java rendszeren teszteltük. Eredetileg 3369 NPE hibajelzés volt ezeken a rendszereken, ezek közül 242-t sikerült eliminálnunk, mint fals pozitív. A kézi ellenőrzés azt mutatta, hogy emellett 7 valódi problémát is felfedeztünk, ami körülbelül 7%-os javulást jelent.

*2. Heurtisztikus javítás egy Java szimbolikus végrehajtó motoron*

A szimbolikus végrehajtás alapja, hogy konkrét értékek helyett a programot szimbolikus értékeken hajtjuk végre. Ha a végrehajtás során feltételes utasításhoz értünk, akkor mind az igaz, mind a hamis végrehajtási útvonal végre lesz hajtva. A feltételes állítások alapján korlátozhatóak a szimbolikus változók értékei. Elméletileg minden lehetséges végrehajtási út be lesz járva, a szimbolikus változók feltételei alapján azonban lesznek olyan utak, amelyeket le lehet metszeni, mivel feltételeik kielégíthetetlenek. A logikai kifejezést, amelyet a szimbolikus változókhoz kapcsolódó korlátozásokból fel lehet építeni, ún. útvonalfeltételnek nevezzük. Az útvonalfeltétel a szimbolikus végrehajtás kulcsfontosságú része, azonban karbantartása és kielégíthetőségének ellenőrzése rendkívül erőforrásigényes feladat. A futási idő nagy részét ez teszi ki, ami miatt a szimbolikus végrehajtók rosszul skálázódhatnak nagyobb rendszereken. Ebben a kutatásban heurisztikus feltétel ellenőrző mechanizmust alkalmaztunk, amelyet null feltétel ellenőrzőnek nevezünk. Ez az ellenőrző csak azt követi nyomon, hogy a szimbolikus Java referenciák értéke null-e. Ez a megoldás nem változtatta meg az RTEHunter számítási időigényét, mivel nincs szükség bonyolult kielégíthetőségi ellenőrzésre. 209 Java rendszeren teszteltük a megoldást. A 3102 NPE-ből 16 súlyos fals pozitív figyelmeztetést elimináltunk és 7 valós problémát is sikerült detektálni.

Ebben a tézispontban olyan módszereket dolgoztunk ki az RTHunter számára, amelyek anélkül növelték annak pontosságát, hogy tovább növelnék a számításigényes szimbolikus végrehajtási folyamat erőforrásigényét. A több, mint kétszáz rendszeren végzett elemzések megerősítették, hogy ezt a célt sikeresen teljesítettük.

***A szerző hozzájárulása***

A szerző elvégezte a két heurisztika elméleti kidolgozását. Irodalmi áttekintést végzett a szimbolikus végrehajtás területén, összegyűjtötte a jelenleg rendelkezésre álló technológiákat és módszertanokat. Leimplementálta a heurisztikákat az RTEHunter rendszerben. Elvégezte a szükséges teszteket kisebb példakódokon, majd több mint 200 Java rendszeren is megvizsgálta a módosítások hatását. Mindkét heurisztika esetében manuálisan validálta az eredményeket. Az értekezéshez kapcsolódó publikációk:

- ♦ **Edit Pengő** and István Siket. Improving Static Initialization Block Handling in Java Symbolic Execution Engine. In Computational Science and Its Applications – ICCSA 2017: 17th International Conference, Proceedings, Part V, pages 561-574. Springer, 2017.
- ♦ **Edit Pengő**. Applying Heuristics to Improve our Java Symbolic Execution Engine – ICAI 2017. In Proceedings of the 10th International Conference on Applied Informatics – ICAI 2017, pages 245-253. Eszterházi Károly Catholic University, 2017.

## II. Statikus hívási gráf számító eszközök összehasonlítása

A második tézispont, mely a Java statikus hívásgráfok vizsgálatához kapcsolódik, az 6 - 9. fejezetekben van tárgyalva.

Mivel a hívási gráfok képezik az interprocedurális elemzések alapját, kiválasztottunk hat nyílt forráskódú statikus elemzőt (SOOT [5], Spoon [78], OSA [5], WALA [6], a JCG [43] és a Soot [85]), hogy összehasonlítsuk az általuk generált hívási gráfokat.

*1. Útmutató Java Hívási Gráfok Összehasonlításához. Találj Párt a Metódusaidnak!*

A hívási gráf generáló eszközök az általuk készített hívási gráfokon keresztül hasonlíthatóak össze. Az általános célú gráfokhoz kifejlesztett módszereket nem lehet közvetlenül alkalmazni a hívási gráfokra, pláne, ha azokat különböző elemző eszközök állították elő. Még ha két hívási gráf felépítése izomorf is, a csomópontok eltérő címkézése miatt teljesen különbözőek lehetnek. Ezért a hívási gráfok összehasonlításához egy megfeleltetést kell készítenünk a csomópontjaik között. Egy heurisztikus algoritmust fejlesztettünk ki több lépésben. Az egész alapja a metódusok nevét, az osztálynevet és a paraméterlistát is tartalmazó teljes elnevezésen alapuló párosítás. Az első lépés az úgynevezett *anonymous transzformáció* volt. Az anonymous forráskód elemeknek nincs szabványosított nevük, ami azt jelenti, hogy a statikus elemzők ugyanazt a kódelemet másképp nevezhetik el. Az anonymous transzformáció egyszerűen azt jelenti, hogy a nevük nem szabványosított részét egy fix karakterlánccal helyettesítjük, ezután a teljes név alapján párosítjuk őket. Ha egy osztályon belül több anonymous osztály is van, akkor ez a megoldás a párosítás pontosságának csökkenését okozza. A következő lépésünk egy *sorszám alapú* párosítás volt. A metódusok nevei mellett a forráskódban elfoglalt helyüket is felhasználtuk a párosításhoz. Sajnos a sorszámozás nem olyan következetes a statikus elemzők körében, mint amire számítottunk, ezért ezt a módszert csak az anonymous és a generikus forráskód elemekre korlátoztuk, míg normál metódusoknál a név szerinti párosítást alkalmaztuk. Végül bevezettünk egy módszert a *Java generikus elemeinek kezelésére*. Ha egy generikus metódust különféle típusokkal példányosítunk, akkor előfordulhat, hogy az egyik eszköz ezt több csomóponttal reprezentálja, míg a másik eszköz csak egy csomóponttal. Ezt a problémát az összes lehetséges párosítás összegyűjtésével oldottuk meg. A sorinformáció alapú és a generikus elemeket kezelő heurisztikák kombinálásával 2-3%-kal javult a csomópont párosítás. Ez jelentős előrelépés, azonban sok csomópont így is párosítatlan maradt. Manuálisan megvizsgáltuk az okokat, ám kiderült, hogy a legtöbb eltérés nem kiküszöbölhető, mivel ezek az eszközök eltérő működéséből adódnak.

*2. Hat nyílt forráskódú Java hívási gráf készítő eszköz szisztematikus összehasonlítása.*

A párosítási algoritmus kidolgozása után elvégeztük a fennmaradó különbségek elemzését. Ezzel az összehasonlítással az volt a célunk, hogy felhívjuk a figyelmet azokra a szempontokra, amelyek alapján kiválasztható a célnak megfelelő hívási gráf készítő eszköz. A hívási gráfok megbízhatósága ugyanis befolyásolhatja a későbbi elemzési folyamatok pontosságát. Lefuttattuk a hat elemzőt egy példakódon, amely a Java 8 nyelvi jellemzőit sűrítette magába, illetve több valós méretű, nyílt forráskódú Java rendszeren, majd elvégeztük az eredmények kvantitatív és kvalitatív elemzését. A különbségek hat forrását azonosítottuk: az inicializáló módszerek, a polimorfizmus, a Java 8 nyelvi elemek (például lambdák), a dinamikus metódus hívások, illetve a generikus és anonymous forráskód elemek eltérő kezelése. Ezek jelentős különbségeket okozhatnak a gráfokban. Habár a párosítási algoritmus nem párosítja az összes lehetséges csomópontot a sorszám információk hiányossága miatt, a különbségek elsősorban nem ebből fakadnak. A kutatás következő lépéseként eltávolítottuk a gráfokból azokat a részeket, amelyek ezekből az ismert különbségekből származnak. Többféle szűrést is kipróbáltunk, végül úgy döntöttünk, hogy csak azokat a metódusokat tartjuk meg a gráfokban, amelyeket minden elemző eszköz megtalált. Az összes többi metódust és a hozzájuk tartozó éleket kiszűrtük, de a gráfok között továbbra is jelentős különbségek maradtak. A manuális vizsgálat feltárt olyan korábban fel nem fedezett tényezőket, mint például az elemzők hibái. Igazoltuk, hogy a hívási gráf készítő eszközök kimenetei az ismert különbségek kiküszöbölésével sem lesznek azonosak.

Ebben a tézispontban megvizsgáltuk a hívási gráfok különbségeinek a mértékét illetve azt is, hogy milyen feldolgozásbeli és a működésbeli különbségek okozzák ezeket a különbségeket. Ez az összehasonlítás segítséget nyújthat a célnak megfelelő hívási gráf készítő eszközök kiválasztásában.

***A szerző hozzájárulása***

A szerző részt vett a hívási gráf készítő eszközök kiválasztásában. A hatból összesen négyért (OSA, WALA, SPOON és JDT) ő volt felelős. Kipróbálta őket, és exportálta belőlük a hívási gráfokat. A szerző kidolgozta a hívási gráf-összehasonlító algoritmus elméleti hátterét. Megtervezte az anonymous átalakítást, a sorszámalapú párosítást és a generikus elemek kezelését. Irodalmi áttekintést végzett a gráf-összehasonlítás területén. A szerző részt vett a megvalósított hívási gráf-összehasonlító program elemzésében és eredményeinek validálásában. A rákövetkező összehasonlító tanulmányhoz frissítette a négy hívási gráf-készítő eszközt és elemezte velük a kiválasztott Java rendszereket. Segített bővíteni a példakódot, amely a Java 8-ra jellemző kódrészleteket tartalmazta. Segített az eredmények értelmezésében és a gráfok közötti különbségek kézi vizsgálatában is. Részt vett a kutatási kérdések kialakításában és a gráfok ismert különbségeinek szűrésére szolgáló lépések elméleti kidolgozásában. Az értekezéshez kapcsolódó publikációk:

- ♦ **Edit Pengő**, Zoltán Ságodi and Ervin Kóbor. Who Are You not gonna Call? A Definitive Comparison of Java Static Call Graph Creator Tools. In The 11th Conference of PhD Students in Computer Science: Volume of short papers – CSCS 2018, pages 68-71. University of Szeged, 2018.
- ♦ Zoltán Ságodi and **Edit Pengő**. A Preparation Guide for Java Call Graph Comparison. Finding a Match for Your Methods. Published in Acta Cybernetica, Volume 24, No 1, Pages 131-155. 2019.
- ♦ Judit Jász, István Siket, **Edit Pengő**, Zoltán Ságodi and Rudolf Ferenc. Systematic Comparison of Six Open-source Java Call Graph Construction Tools. In Proceedings of the 14th International Conference on Software Technologies – ICSOFT 2019, pages 117-128. SciTePress, 2019.

## III. Primitive Obsession metrikák kidolgozása és kiértékelése

A harmadik tézispont egy eddig kevésbé tanulmányozott gyanús kóddal, a Primitive Obsessionnel foglalkozik. A 10 - 13. fejezetek tárgyalják.

*1. A Primitive Enthusiasm megragadása - a Primitive Obsession megközelítése lépésekben.* Egy előzetes tanulmányban megalkottuk a Primitive Enthusiasm (PE) nevű metrikát, amely segít kimutatni a primitív típusok túlzott használatát a metódusok paraméterlistáiban. Ebben a munkában új metrikákat vezettünk be a Primitive Enthusiasm további aspektusainak leírására. Először átneveztük a Primitive Enthusiasm-ot *Local Primitive Enthusiasm-ra (LPE)*, azután további variánsokat vezettünk be: *Global Primitive Enthusiasm (GPE)*, amely egy globális küszöbhöz viszonyítva észleli a primitív paraméterek túlzott mértékű használatát és a *Hot Primitive Enthusiasm (HPE)*, amely csak azokat a metódusokat jelenti, amelyeket mindkét korábbi PE változat megjelölt. Megpróbáltuk jellemezni a gyanús kód egyéb előfordulási formáit is. A típuskódok használatát a *Static Final Primitives (SFP)* és a *Static Final Primitives - Switch Case Usage (SFP-SCU)* metrika észleli. A *Method Parameter Clones (MPC)* megjelöli azokat a metódus paramétereket, amelyeket ki lehetne emelni egy értékobjektumba. Ezt a hat metrikát leimplementáltuk az OpenStaticAnalyzer (OSA) [5] statikus elemzőben, és három valós méretű Java-projekten teszteltük. Két metódus eliminálási stratégiával kísérleteztünk a pontosabb eredmények elérése érdekében. A kísérlet eredményeképpen minden egy vagy kevesebb paraméterrel rendelkező metódust kihagytunk a számításból. A gyanúsnak jelzett metódusokat, osztályokat mintavételeztük és manuálisan kivizsgáltuk. A megjelölt metódusok száma átlagosan kevesebb, mint az összes metódus 8%-a. Az *MPC* metrika felhasználható lehet ezen eredmények súlyozásához. Az *SFP-SCU* metrika több típuskód-használatot észlelt a projektekben. Az eredmények azt mutatták, hogy az új metrikák illetve azok kombinációi sok gyanús és nehezen olvasható kódszegmenst emelnek ki.

*2. A Primitive Obsession metrikák hibaelőrejelzési képességeinek vizsgálata.*

Ebben a kutatásban integráltuk a három osztályszintű Primitive Obession metrikát (az *MPC* , az *SFP* és az *SFP-SCU* metrikákat) egy meglévő hibaelőrejelzési adatbázisba [37], és megvizsgáltuk ennek a hatását a J48 algoritmus segítségével. Az eredeti illetve bővített adathalmazok hibaelőrejelzési képességét a J48 algoritmussal teszteltük. Az F-mérték[1] stagnálást vagy enyhe csökkenést mutatott. Arra a következtetésre jutottunk, hogy a PO metrikák hozzáadása az eredeti adatkészlethez nem javítja a hibaelőrejelzési képességet. A projektek közötti validálás azonban más képet mutatott. Minden projektre betanítottunk egy modellt és kiértékeltük azt az összes többi projekten. Összesen 29 rendszert használtunk fel, tehát minden modellt 28 rendszeren értékeltünk ki. A betanításokat elvégeztük az eredeti adathalmazon és a PO metrikákkal bővített adathalmazon is és megvizsgáltuk az eredmények különbségeit. 841 esetből az F-mérték 194 esetben csökkent és 240 esetben javult. Összességében elmondható, hogy a PO metrikák hozzáadása több javulást eredményez, mint romlást a projektek közötti validáció esetén, de fontos, hogy a modell építéséhez megfelelő rendszert válasszunk ki. Kiszámítottuk a korrelációs mátrixot [2] is illetve főkomponens-analízist (PCA) [89] hajtottunk végre a PO-metrikák dimenzionalitásának vizsgálatához. A korreláció nem mutatott szignifikáns lineáris kapcsolatot sem a PO-metrikák között, sem a PO-metrikák és a többi metrika között. A PCA-számításból kiderült, hogy az adathalmaz varianciájának 99%-ához 33 fő komponensre van szükség, ami az eredeti attribútumok több mint fele. Még 95% is 20 komponenst igényel. Kiszámítottuk a faktorsúlyokat [3] erre a 20 főkomponensre. Az eredmények azt mutatták, hogy a PO mérőszámok háromhoz is jelentősen hozzájárulnak.

Ebben a tézispontban megmutattuk, hogy a Primitive Obession metrikák egy új, hasznos megközelítést nyújtanak a forráskód jellemzésére és mérésére, sőt, képesek lehetnek javítani a hibaelőrejelzésen is.

### *A szerző hozzájárulása*

A szerző segített az elemzett rendszerek kiválasztásában és az eredeti Primitive Enthusiasm metrika kiértékelésében. Irodalmi áttekintést végzett a gyanús kódok és a Primitive Obsession területén. Megtervezte az *MPC*, az *SFP* és az *SFP-SCU* metrikákat. Az előzetes tanulmány továbbfejlesztéséhez újraimplementálta a Primitive Enthusiasm számítását és elvégezte a másik öt metrika implementációját

[1]Az F-mérték a precizitás és a felidézés harmonikus középértéke.

[2]Kísérletünkhöz Pearson-féle korrelációt használtunk.

[3]A faktorsúly megmutatja, hogy az eredeti attribútumok mennyiben járulnak hozzá az adott főkomponenshez.

is az OSA statikus elemzőben. Kísérletezett a metódus eliminációs technikákkal, ez alapján ő választotta ki a kutatásban felhasznált eliminációs módszert. Részt vett az elemzett Java rendszerek kiválasztásában és az eredmények kiértékelésében is. Megvizsgálta a metrikák eredményeinek átfedéseit és azok megfelelését. A szerző implementált egy Weka-alapú alkalmazást a hiba-adatbázissal kapcsolatos kutatáshoz. Kiszámította 58 rendszeren a Primitive Obsession metrikákat és integrálta az adatokat egy létező hiba-adatbázisba. Készített egy kiértékelő programot az eredeti és a kibővített adatsor eredményeinek összehasonlítására. Megfogalmazta a kutatási kérdéseket és elvégezte az eredmények kiértékelését és értelmezését. A tézisponthoz kapcsolódó publikációk:

- Péter Gál and **Edit Pengő**. Primitive Enthusiasm: A Road to Primitive Obsession. In The 11th Conference of PhD Students in Computer Science: Volume of short papers – CSCS 2018, pages 134-137. University of Szeged, 2018.
- **Edit Pengő** and Péter Gál. Grasping Primitive Enthusiasm - Approaching Primitive Obsession in Steps. In Proceedings of the 13th International Conference on Software Technologies – ICSOFT 2018, pages 389-396. SciTePress, 2018.
- **Edit Pengő**. Examining the Bug Prediction Capabilities of Primitive Obsession Metrics. In Computational Science and Its Applications – ICCSA 2021: 21st International Conference, Proceedings, Part VII, pages 185-200. Springer, 2021.

A tézispontokat és a kapcsolódó publikációkat a B.1. táblázat összegzi.

| № | [118] | [119] | [116] | [115] | [113] | [112] | [114] | [117] |
|---|---|---|---|---|---|---|---|---|
| I. | ♦ | ♦ | | | | | | |
| II. | | | ♦ | ♦ | ♦ | | | |
| III. | | | | | | ♦ | ♦ | ♦ |

**B.1. táblázat.** A tézispontokhoz kapcsolódó publikációk

# Acknowledgement

- This thesis was partially supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled Internet of Living Things.
- This work was supported by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).
- Part of the results of this dissertation were obtained in the SETIT Project (2018-1.2.1-NKP-2018-00004) Project no. 2018-1.2.1-NKP-2018-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.
- The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (MILAB).

# Bibliography

[1] Apache BCEL Home Page.
`https://commons.apache.org/proper/commons-bcel`. (Accessed: 2022-05-17).

[2] Eclipse Home Page.
`www.eclipse.org/eclipse/`. (Accessed: 2022-05-17).

[3] Eclipse JDT Home Page.
`http://www.eclipse.org/jdt/`. (Accessed: 2022-05-17).

[4] Java PathFinder Tool-set. `https://github.com/javapathfinder/jpf-core`. (Accessed: 2022-05-17).

[5] Open Static Analyser GitHub Page.
`https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer`. (Accessed: 2022-05-17).

[6] WALA Home Page.
`http://wala.sourceforge.net/wiki/index.php/Main_Page`. (Accessed: 2022-05-17).

[7] Cyberpunk 2077 disaster.
`https://www.bloomberg.com/news/articles/2020-12-07/cd-projekt-s-cyberpunk-game-is-a-critical-hit-despite-glitches`, 2021. (Accessed: 2022-05-17).

[8] Sable *J Home Page.
`http://www.sable.mcgill.ca/starj/`, 2022. (Accessed: 2022-05-17).

[9] Laura A. Zager and George Verghese. Graph similarity scoring and matching. *Applied Mathematics Letters*, 21(1):86–94, 2008.

[10] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 688–712. Springer-Verlag, 2012.

[11] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[12] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, 1994.

[13] Gábor Antal, Péter Hegedűs, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. 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, 2018.

[14] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. *Commun. ACM*, 59(6):93–100, 2016.

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

[16] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *arXiv preprint arXiv:1610.00502*, 2016.

[17] Brett Becker and Catherine Mooney. Categorizing compiler error messages with principal component analysis. In *Proceedings of the 12th China-Europe International Symposium on Software Engineering Education (CEISEE '16)*, pages 1–8, 2016.

[18] Dirk Beyer. Advances in automatic software verification: Sv-comp 2020. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 347–367. Springer International Publishing, 2020.

[19] Sajad Ahmad Bhat and Jatender Singh. A Practical and Comparative Study of Call Graph Construction Algorithms. *IOSR Journal of Computer Engineering*, 1(4):14–26, 2012.

[20] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

[21] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245. ACM, 1975.

[22] Cristian Cadar, Daniel Dunbar, Dawson R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[23] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335. ACM, 2006.

[24] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[25] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346 – 7354, 2009.

[26] Pierre-Antoine Champin and Christine Solnon. Measuring the similarity of labeled graphs. In *Case-Based Reasoning Research and Development*, pages 80–95, 2003.

[27] Laila Cheikhi, Rafa Al-Qutaish, Ali Idri, and Asma Sellami. Chidamber and kemerer object-oriented measures: Analysis of their design from the metrology perspective. *International Journal of Software Engineering and Its Applications*, 8:359–374, 03 2014.

[28] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 169–186. USENIX Association, 2003.

[29] P. David Coward. Symbolic Execution Systems – a Review. *Software Engineering Journal*, 3(6):229–239, November 1988.

[30] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41, 2010.

[31] Danny van Bruggen, Federico Tomassetti, Nicholas Smith, Cruz Maximilien. JavaParser - for processing Java code Homepage. `https://javaparser.org/`. (Accessed: 2022-05-17).

[32] 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, August 7–11, 1995*, pages 77–101. Springer Berlin Heidelberg, 1995.

[33] 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. Springer Berlin Heidelberg, 2008.

[34] M. A. Eshera and K. Fu. A graph distance measure for image analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-14(3):398–408, May 1984.

[35] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 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. ACM, 2014.

[36] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, IEEE Computer Society, oct 2002.

[37] Rudolf Ferenc, Zoltán Tóth, Gergely Ladányi, István Siket, and Tibor Gyimóthy. A public unified bug dataset for java and its assessment regarding metrics and bug prediction. *Software Quality Journal*, 2020.

[38] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello. An experience report on using code smells detection tools. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 450–457. IEEE, 2011.

[39] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[40] Eibe Frank, Mark A. Hall, and Ian H. Witten. *Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, Fourth Edition, 2016.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[42] Gecode Homepage. `https://www.gecode.org`. (Accessed: 2022-05-17).

[43] Georgios Gousios. Java Call Graph GitHub Page. `https://github.com/gousiosg/java-callgraph`. (Accessed: 2022-05-17).

[44] A. Gerasimov, S. Vartanov, M. Ermakov, L. Kruglov, D. Kutz, A. Novikov, and S. Asryan. Anxiety: A dynamic symbolic execution framework. In *2017 Ivannikov ISPRAS Open Conference (ISPRAS)*, pages 16–21, 2017.

[45] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005.

[46] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2008.

[47] K.R. Godfrey. Correlation methods. *IFAC Proceedings Volumes*, 12:527–534, 1979. Tutorials presented at the 5th IFAC Symposium on Identification and System Parameter Estimation, Darmstadt, Germany, September.

[48] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[49] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 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. ACM, 1997.

[50] Aakanshi Gupta, Bharti Suri, and Sanjay Misra. A systematic literature review: Code bad smells in java source code. In *Computational Science and Its Applications – ICCSA 2017*, pages 665–682. Springer, 2017.

[51] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[52] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology*, 23(4), 2014.

[53] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin T. Vechev. Learning to explore paths for symbolic execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2526–2540. ACM, 2021.

[54] Hessel Hoogendorp. Extraction and visual exploration of call graphs for Large Software Systems. Master's thesis, University of Groningen, 2010.

[55] Yue Jiang, Bojan Cukic, and Yan Ma. Techniques for evaluating fault prediction models. *Empirical Softw. Engg.*, 13(5):561–595, 2008.

[56] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10. ACM, 2010.

[57] István Kádár, Péter Hegedűs, and Rudolf Ferenc. Runtime exception detection in java programs using symbolic execution. *Acta Cybernetica*, 21(3):331–352, 2014.

[58] István Kádár, Péter Hegedűs, and Rudolf Ferenc. Adding constraint building mechanisms to a symbolic execution engine developed for detecting runtime errors. In *In Proceedings of the 15th International Conference on Computational Science and Its Applications*, pages 20–35, 2015.

[59] Arvinder Kaur and Inderpreet Kaur. An empirical evaluation of classification algorithms for fault prediction in open source projects. *Journal of King Saud University - Computer and Information Sciences*, 30, 2016.

[60] F. Khomh, M. Di Penta, and Y. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84, 2009.

[61] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[62] Sebastian Kloibhofer, Thomas Pointhuber, Maximilian Heisinger, Hanspeter Mössenböck, Lukas Stadler, and David Leopoldseder. Symjex: Symbolic execution on the graalvm. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, MPLR 2020, page 63–72. Association for Computing Machinery, 2020.

[63] Danai Koutra, A Parikh, A Ramdas, and J Xiang. Algorithms for Graph Similarity and Subgraph Matching. Technical report, 2011.

[64] Ondrej Lhoták. Comparing call graphs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42, 2007.

[65] Ondřej Lhoták and Laurie Hendren. Context-Sensitive Points-to Analysis: Is It Worth It? In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, pages 47–64. Springer Berlin Heidelberg, 2006.

[66] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S. Yu. Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. In *Proceedings of the 2005 SIAM International Conference on Data Mining (SDM)*, pages 286–297, 2005.

[67] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman.

Jdart: A dynamic symbolic analysis framework. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 442–459. Springer Berlin Heidelberg, 2016.

[68] O. Macindoe and W. Richards. Graph comparison using fine structure analysis. In *2010 IEEE Second International Conference on Social Computing*, pages 193–200, Aug 2010.

[69] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings - International Conference on Software Engineering*, pages 416–425, 2007.

[70] Mika V. Mäntylä, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance. ICSM*, pages 381–384. IEEE, 2003.

[71] Mika V. Mäntylä, Jari Vanhanen, and Casper Lassenius. Bad smells - humans as code critics. In *Proceedings of the 20th IEEE International Conference on Software Maintenance. ICSM*, pages 399–408. IEEE, 2004.

[72] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayse Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, 17:375–407, 2010.

[73] Leon Moonen and Aiko Yamashita. Do code smells reflect important maintainability aspects? In *Proceedings of the 2012 IEEE International Conference on Software Maintenance. ICSM*, pages 306–315. IEEE, 2012.

[74] Malte Mues and Falk Howar. Jdart: Dynamic symbolic execution for java bytecode (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 398–402. Springer International Publishing, 2020.

[75] 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, 1998.

[76] Ciprian Paduraru, Bogdan Ghimis, and Alin Stefanescu. Riverconc: An open-source concolic execution engine for x86 binaries. In *Proceedings of the 15th International Conference on Software Technologies, ICSOFT*, pages 529–536. ScitePress, 2020.

[77] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, 45(2):194–218, 2019.

[78] 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.

[79] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180. ACM, 2010.

[80] Qinbao Song, M. Shepperd, M. Cartwright, and C. Mair. Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering*, 32(2):69–82, 2006.

[81] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64. USENIX Association, 2015.

[82] 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. ACM, 2016.

[83] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 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. ACM, 2018.

[84] Naveen Roperia. Jsmell: A bad smell detection tool for java systems. Master's thesis, Maharishi Dayanand University, 2009.

[85] Sable Research Group. Sable/Soot GitHub Page. `https://github.com/Sable/soot`. (Accessed: 2022-05-17).

[86] A. Sanfeliu and K. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3):353–362, May 1983.

[87] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423. Springer-Verlag, 2006.

[88] Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser. Java ranger: Statically summarizing regions for efficient symbolic execution of java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 123–134. Association for Computing Machinery, 2020.

[89] Jonathon Shlens. A tutorial on principal component analysis, 2014.

[90] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.

[91] The SourceMeter Home Page. `https://www.sourcemeter.com`. (Accessed: 2022-05-17).

[92] Matheus Souza, Mateus Borges, Marcelo D'Amorim, and Corina S. Păsăreanu. CORAL: Solving complex constraints for symbolic pathfinder. In *Lecture Notes in Computer Science*, volume 6617 LNCS, pages 359–374. Springer, Berlin, Heidelberg, 2011.

[93] 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, 2000.

[94] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, 2010.

[95] Nikolai Tillmann and Jonathan De Halleux. Pex: White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153. Springer-Verlag, 2008.

[96] Frank Tip and Jens Palsberg. 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. ACM, 2000.

[97] Zoltán Tóth, Péter Gyimesi, and Rudolf Ferenc. A public bug database of github projects and its application in bug prediction. In *Computational Science and Its Applications – ICCSA 2016*, volume 9789, pages 625–638. Springer, 2016.

[98] Amos Tversky. Features of similarity. *Psychology Review*, 84:327–352, 07 1977.

[99] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[100] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. *SIGPLAN Not.*, 29(6):85–96, 1994.

[101] R. S. Wahono. A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks. *Journal of Software Engineering*, 1:1–16, 2015.

[102] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449. IEEE Press, 1981.

[103] Elaine Weyuker, Thomas Ostrand, and Robert Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15:277–295, 2010.

[104] Nicolas Wicker, Canh Hao Nguyen, and Hiroshi Mamitsuka. A new dissimilarity measure for comparing labeled graphs. *Linear Algebra and its Applications*, 438(5):2331 – 2338, 2013.

[105] A. Yamashita and L. Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 682–691, 2013.

[106] Aiko Yamashita and Leon Moonen. To what extent can maintenance problems be predicted by code smell detection? - An empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013.

[107] Zhifeng Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 293–299, 2001.

[108] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: A review of current knowledge. *Journal of Software Maintenance and Evolution*, 23(3):179–202, 2011.

[109] Y. Zhang, Z. Chen, Z. Shuai, T. Zhang, K. Li, and J. Wang. Multiplex symbolic execution: Exploring multiple paths by solving once. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 846–857, 2020.

[110] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 91–100. Association for Computing Machinery, 2009.

[111] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, page 9. IEEE, 2007.

## Corresponding Publications of the Author

[112] Péter Gál and Edit Pengő. Primitive enthusiasm: A road to primitive obsession. In *The 11h Conference of PhD Students in Computer Science: Volume of short papers – CSCS 2018*, pages 134–137. University of Szeged, 2018.

[113] Judit Jász, István Siket, Edit Pengo, Zoltán Ságodi, and Rudolf Ferenc. Systematic comparison of six open-source java call graph construction tools. In *Proceedings of the 14th International Conference on Software and Data Technologies – ICSOFT 2019*, pages 117–128. SciTePress, 2019.

[114] Edit Pengő and Péter Gál. Grasping primitive enthusiasm - approaching primitive obsession in steps. In *Proceedings of the 13th International Conference on Software Technologies – ICSOFT 2018*, pages 389–396. SciTePress, 2018.

[115] Edit Pengő and Zoltán Ságodi. A preparation guide for java call graph comparison: Finding a match for your methods. *Acta Cybernetica*, 24(1):131–155, 2019.

[116] Edit Pengő, Zoltán Ságodi, and Ervin Kóbor. Who are you not gonna call? a definitive comparison of java static call graph creator tools. In *The 11th Conference of PhD Students in Computer Science: Volume of short papers – CSCS 2018*, pages 68–71. University of Szeged, 2018.

[117] Edit Pengő. Examining the bug prediction capabilities of primitive obsession metrics. In *Computational Science and Its Applications – ICCSA 2021: 21st International Conference, Proceedings, Part VII*, pages 185–200. Springer, 2021.

[118] Edit Pengő and István Siket. Improving static initialization block handling in java symbolic execution engine. In *Computational Science and Its Applications – ICCSA 2017: 17th International Conference, Proceedings, Part V*, pages 561–574. Springer, 2017.

[119] Edit Pengő. Applying heuristics to improve our java symbolic execution engine. In *Proceedings of the 10th International Conference on Applied Informatics – ICAI 2017*, pages 245–253. Eszterházy Károly Catholic University, 2017.