API Finder: Accurate Extraction of Method Binding Information from Call Sites without Building the Code

Diptopol Dam

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

May 2023

© Diptopol Dam, 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Diptopol Dam**

Entitled: **API Finder: Accurate Extraction of Method Binding Information from Call Sites without Building the Code**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________________ Chair
Dr. Tse-Hsun (Peter) Chen

______________________________ Examiner
Dr. Joey Paquet

______________________________ Supervisor
Dr. Nikolaos Tsantalis

Approved by ______________________________
Dr. Leila Kosseim, Graduate Program Director

__________ 2023

____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

API Finder: Accurate Extraction of Method Binding Information from Call Sites without Building the Code

Diptopol Dam

Researchers and practitioners have introduced many static analysis tools to enhance the quality of software. Several tools (e.g., WALA [1]) are highly effective and based on mature compilers. However, all of these tools usually require a complete codebase to compile the project under analysis. In many scenarios, academics and practitioners need to analyze partial programs collected from the web or online source code repositories. A partial program is a subset of the complete codebase. The compiler requires a complete codebase to resolve all binding information to identify type information of program elements (e.g., the return type and the argument types of method invocation, type of field instance), and as a result, most of the static analysis tools cannot extract type information for program elements in partial programs.

In this paper, we introduce API Finder, a tool that can accurately extract method binding information from method references (i.e., call sites) in partial programs. Our approach requires only the Java version of the project, the dependent external library artifacts, and the method invocation as inputs in order to generate precise method-binding information. We also provide support for extracting the Java version of the project, as well as dependent external library artifacts for the Gradle and Maven build systems. We evaluated the accuracy of the method-binding information generated by our tool with the Eclipse JDT (Java Development Tool) Compiler across eleven complete projects. Our tool has an accuracy rate ranging between 93% to 99% in our evaluated projects. Our tool also has an average response time of 381 milliseconds for any method-binding information extraction. In addition, as an application of our tool, we have implemented a Chrome browser extension for displaying the method signature upon clicking on any method reference for the GitHub platform.

[1]https://wala.sourceforge.net

Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Nikolaos Tsantalis. In the perusal of my research, his guidance and support were invaluable. His feedback and direction assisted me in overcoming the challenges encountered during my research.

In addition, I would like to thank my committee members, Dr. Joey Paquet and Dr. Tse-Hsun (Peter) Chen, for allocating their valuable time to reviewing the thesis work.

Finally, I would also like to thank my colleagues for sharing their research experiences and giving me the opportunity to gain knowledge.

Thank you.

Diptopol Dam

Contents

List of Figures **vii**

List of Tables **viii**

1 Introduction **1**

1.1 Motivation 2

1.1.1 Practitioner Perspective 2

1.1.2 Researcher Perspective 3

1.1.3 Limitations of Existing Tools 5

1.2 Contribution 5

2 Literature Review **7**

2.1 Partial Program Analysis 7

2.2 Limitation of the existing approaches 9

3 Background **12**

4 Approach **19**

4.1 Artifact & Java Version Extraction 19

4.1.1 Input 20

4.1.2 Determination of Build System 20

4.1.3 Extraction of Java Version & Artifacts 22

4.1.4 Extraction Of Java Core Packages 23

4.1.5 Extraction of Artifact Archives 23
4.1.6 Storing process 23
4.1.7 Output 24
4.2 Method Binding Information Resolution 24
4.2.1 Invocation context information extraction 24
4.2.2 Resolution of Appropriate Method Binding Information 34
4.2.3 Post-Processing of Method Binding Information: 64
4.2.4 Conversion of variadic Argument: 67

5 Implementation **68**
5.1 "API Finder" API 68
5.2 "API Finder" API Using Fluent Builder Pattern 71
5.3 Storage 74
5.4 Chrome Extension for GitHub 76
5.4.1 Comparison between "API Finder" extension and GitHub Code Navigation 77

6 Evaluation **80**
6.1 Evaluation Setup 80
6.1.1 Project Selection 80
6.1.2 Evaluation Process 82
6.2 RQ 1: What is the accuracy of our proposed approach? 83
6.3 What is the execution time for resolving method binding information? 92
6.4 Comparison between "API-Finder" and "JavaSymbolSolver" 98
6.5 Limitations and Threats to Validity 100

7 Conclusion and Future Work **103**
7.1 Applications of "API Finder" 104
7.2 Future Work 104

Bibliography **105**

List of Figures

Figure 1.1 Diff View Collected From a Pull Request of JFreeChart[2] 2

Figure 1.2 GROUM Representation 4

Figure 3.1 A Simple Java Program with the corresponding AST 13

Figure 3.2 POM File 15

Figure 3.3 Gradle Build File 17

Figure 4.1 Overview of our approach 20

Figure 4.2 Overview of Java Version & Artifact Extraction 21

Figure 4.3 Overview of Method Binding Information Resolution 25

Figure 5.1 Entity Relationship Diagram 75

Figure 5.2 "API Finder" Chrome Extension (Collected from JFreeChart [3]) 76

Figure 5.3 No Consideration of Invocation Context (Collected from JFreeChart [4]) . . . 78

Figure 5.4 No Consideration of Number of Method Arguments (Collected from JFreeChart[5]) 78

Figure 6.1 Box-plot of Execution Time for Projects 95

Figure 6.2 Median of execution Time over Resolved Method Count 97

Figure 6.3 Execution Time for Internal and External Method Invocation 98

Figure 6.4 Execution Time for Method Invocation Expressions 99

List of Tables

Table 3.1 AST Node types for Method Invocation Expression 13

Table 4.1 Type Representation 28

Table 4.2 Eligible Widening Primitive Type Conversions 52

Table 4.3 Eligible Narrowing Primitive Type Conversions 52

Table 4.4 Primitive Types and their corresponding Wrapper Class 54

Table 4.5 Conversion Type and their corresponding Distance 63

Table 4.6 Type With Generic Signature 65

Table 5.1 API for Loading Project Java and Dependent Artifacts [*TypeInferenceAPI* Class] 69

Table 5.2 "API Finder" API [*TypeInferenceV2API* Class] 69

Table 5.3 "API Finder" API Using Fluent Builder Pattern [*TypeInferenceFluentAPI* Class] 72

Table 6.1 Java Projects for Evaluation 82

Table 6.2 Distribution of Method Invocation Type 84

Table 6.3 Evaluation of Accuracy 85

Table 6.4 Mismatch Distribution 86

Table 6.5 Execution Time 96

Table 6.6 Distribution of execution time for Number of Resolved Methods 96

Table 6.7 Accuracy comparison between API-Finder and JavaSymbolSolver 100

Chapter 1

Introduction

Static program analysis plays a vital role in software development and software engineering research. Several static analysis tools help developers with the optimization of the programs and maintaining standard code conventions in the software development process. In addition, they are heavily used for the detection of feature location [1], bug detection [2], analysis of external API usage [3] research.

There are several situations that require partial program analysis [4], including mining bugfixes for automatic patching [5], defect prediction [6], analyzing code fragments from forum threads for recommendation [7] and ranking code search results [8].

Binding Information connects references to types, methods, fields, and variables to their actual declaration (i.e., type, method, field, variable declarations). Leveraging the connection between reference and declaration, binding information can provide a few key valuable pieces of information regarding the particular type, method, or field (e.g., name of the class where the method is declared, types of the method argument, type parameters of any parameterized argument type). A compiler can extract binding information after building the complete codebase. However, a compiler cannot generate binding information for partial programs.

```
343 +
344 +        // get the data points
345 +        double x1 = dataset.getXValue(series, item);
346 +        double y1 = dataset.getYValue(series, item);
347 +        double transX1 = xAxis.valueToJava2D(x1, dataArea, xAxisLocation);
348 +        double transY1 = yAxis.valueToJava2D(y1, dataArea, yAxisLocation);
```

Figure 1.1: Diff View Collected From a Pull Request of JFreeChart[1]

1.1 Motivation

1.1.1 Practitioner Perspective

We wish to illustrate a motivating scenario in which a software developer is performing the task of code review for a pull request. Figure 1.1 shows a portion of the change set written in Java programming language, that is part of a pull request. During the review process, the developer may want to know additional information which will be useful for the code review. Information such as:

- The class name of the invoked method
- Type of the arguments of the invoked method
- The nature of invoked method (e.g., whether belongs to external libraries, nature of the external libraries)

Extraction of method binding information can provide us with such information. A Java compiler can generate method binding information for any method invocation expression. However, there are a few challenges that restrict us to use compilers to generate method binding information for this scenario. The challenges are:

- The change set is partial and the access to the complete code base may not be accessible. However, the compiler will require the access to complete code base in order to build and generate the method binding information.
- The change set may not be buildable which restricts the compiler to generate the method binding information.

[1] https://github.com/jfree/jfreechart/pull/286/commits/f4b5bc7cd5ea6b44a36ce57d0a1f19f5786839a2

- To access the complete code base, we may need to clone the project and make the project accessible to the compiler. However, cloning a project is a time-consuming task for large projects. To ensure the usability of this feature, we need to identify method binding information in a quick manner. Therefore, we cannot rely upon the compiler to generate the method binding information.

GitHub has implemented symbolic code navigation [9] for the GitHub platform where users can click on any method invocation expression and GitHub will present a list of probable method declarations in a tooltip. Users can navigate through the declarations of the method. However, the approach has several limitations. The limitations are:

- Code navigation does not show method declaration instances that are outside of the project.
- To show appropriate method declaration, they do not take consideration of types of arguments, or type of the method invoker expression.

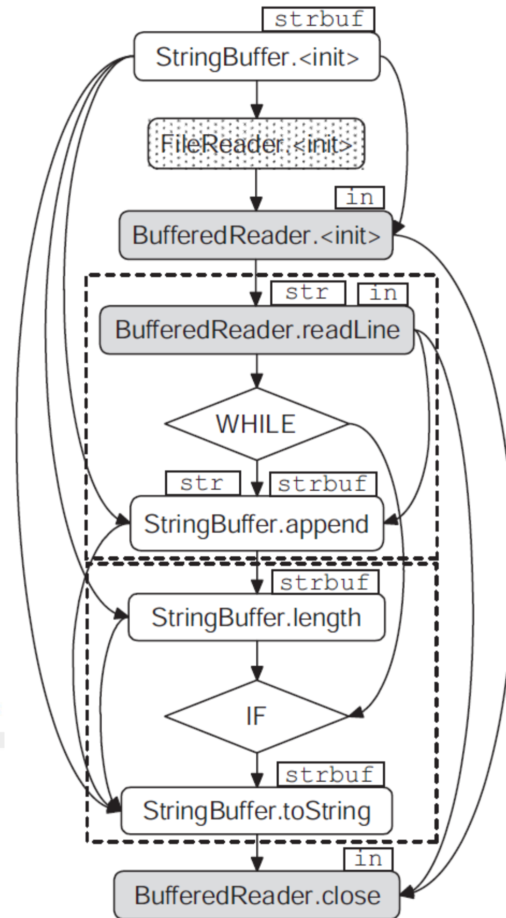
We have provided a more in-detail explanation of all the limitations in Section 5.4.1.

1.1.2 Researcher Perspective

Many researchers often perform large-scale empirical studies on a large number of commits. In many cases, they may need to extract different representations from the source code. GROUM [10] is such a representation, which is very useful to recognize patterns with control and conditional structure that can be helpful for anomaly detection, as well as source code pattern recognition. Figure 1.2 shows the GROUM representation of a code snippet (Figure 1.2a). In Figure 1.2b, each invoked method is represented as a node and the representation requires the class name where the method is declared. Using method binding information, we can generate such a representation. However, a recent study [11] has shown that only 38% of the change history of software systems can be successfully compiled, and thus we cannot rely on the compiler to generate the binding information.

```
1  public void method() {
2    StringBuffer strbuf = new StringBuffer();
3    BufferedReader in =new BufferedReader(new FileReader(file));
4
5    String str;
6
7    while ((str = in.readLine()) != null) {
8      strbuf.append(str + "\n");
9    }
10
11   if (strbuf.length() > 0) {
12     outputMessage(strbuf.toString());
13   }
14
15   in.close();
16 }
```

(a) Code Snippet



(b) GROUM Representation [10]

Figure 1.2: GROUM Representation

1.1.3 Limitations of Existing Tools

Dagenais et al. [4] have proposed a framework named PPA that attempts to tackle this challenge of identifying type information via heuristics from only the set of partial sources that are available. Hao Zhong et al. [12] also introduced an approach named GRAPA to improve the inference strategies of PPA and make partial programs viable for the static analysis tool WALA. However, there are several limitations.

- The inability to support new Java language functionalities (i.e., consideration of lambda expressions, generics).
- The reliance of GRAPA on the existing complete code analysis tools, such as WALA makes it tightly coupled with complete source code analysis tools making it difficult for other practitioners to implement for their own system.

More details about the limitations of these tools are provided in Section 2.2.

1.2 Contribution

In this research paper, we present "API Finder", an accurate extraction of method binding information from method invocation expressions in a partial program. Binding is the information that connects references to types, methods, fields, and variables to their actual declaration (i.e., type, method, field, and variable declarations). We present a streamlined approach to generating binding information for any method invocation expression. We performed an evaluation of our approach against binding information generated by the compiler. Our approach has performed with higher accuracy (above 93%) and the execution time for resolving each method invocation expression has been on average 380 milliseconds.

This thesis provides the following major contributions:

- Our approach provides a streamlined API for extracting appropriate method-binding information from any method invocation expression. We process the new language syntax (i.e., generics, lambda expressions) to generate accurate method binding information. We support

Java language versions up to Java SE 11. Our API implementation will help practitioners easily integrate the approach into their systems.

- We facilitate the extraction of dependent artifacts from project build files. Gradle and Maven are two of the most popular build systems for Java projects. Our approach supports parsing build files from both build systems to collect all dependent external library artifacts and determine the Java version of the source code.

- We have developed a Chrome extension for displaying declared method signatures from any invocation expression on GitHub. In several scenarios, we have outperformed the suggestions provided by the state-of-the-art source code navigation of GitHub for the Java programming language.

The rest of the thesis is organized as follows: The second chapter provides a related literature review on partial program analysis. Chapter 3 provides the background information required to understand the rest of the thesis. In Chapter 4, we explain our approach to identifying and extracting accurate method-binding information. In Chapter 5, we discuss our implementation. Chapter 6 describes the evaluation process for our methodology. Finally, in Chapter 7, we present our conclusion and discuss potential future research directions.

Chapter 2

Literature Review

This section will begin with a discussion of studies relevant to partial program analysis and identifying program elements from partial programs. The limitations of these existing studies will next be discussed.

2.1 Partial Program Analysis

Thummalapenta et al. [13] have proposed an approach to suggest relevant method invocation sequences for converting a source object to a destination object for existing frameworks and libraries. Identifying all the types associated with method invocation statements (e.g., types of arguments, return type) from partial programs is one of the problems that they tried to solve in their methodology. They proposed a few heuristics for identifying type information. Five heuristics have been introduced to resolve method invoker object type and method argument types. Additionally, ten heuristics have been introduced to identify the return type of the method. However, only two major heuristics for identifying the type of return object were explained in the paper.

- In an initialization expression, the return type of a method invocation statement will be the same type as the declared variable.
- The return type of a method invocation that resides on a return statement is the same as the return type of the method declaration that encloses it.

Dagenais et al. [4] have also presented a framework named Partial Program Analysis [PPA] to perform partial type inference and have used heuristics to resolve syntactic ambiguities to recover the declared types of the expressions. The framework produces a complete typed Intermediate Representation (IR). The authors conducted an empirical study for the framework on four large open-source projects, which shows that the framework recovers most of the declared types, with 91.2% correct types and 2.7% erroneous types where only one class is accessible. The authors have listed 11 inference strategies for determining the type of the expression and identifying appropriate method bindings.

Zhong et al. [12] have presented a general approach named GRAPA where they resolve the missing binding information or unknown code names for partial programs using the project's released archivables and various inference strategies. The authors view released archivable as a complete context of the program. At first, they try to identify the release version of the archivable where the partial program may belong. To accomplish that task they try to look for all the code names that are available in the partial programs in the different release versions. As the next step, they compile the partial program in the context of release archivable. Resolving all unknown bindings has allowed them to compile the partial program and makes the partial program ready to be analyzed by the static analysis tool WALA, which is usually used for analyzing complete programs. They implemented a tool leveraging WALA to generate a system dependency graph for partial programs. To demonstrate the accuracy of GRAPA, they performed an evaluation on 8,198 partial commits from four popular open-source projects. GRAPA has successfully resolved unknown bindings for 98.5% of bug fixes with a rate of accuracy of 96.1%.

Gagnon et al. [14] have presented a type inference algorithm for byte code representation. They have performed type resolution for the local variables in byte code representation. They have modeled the problem of type inference into a constraint system represented as a directed graph. The graph consists of the hard node which represents an explicit type, the soft node which represents a type variable, and a direct edge to represents constraint between nodes. In three stages, the algorithm attempts to merge intermediate soft nodes into hard nodes using the assignment inference strategy and tries to reduce the redundant transitive constraint edges until all types are resolved given that all the required constraints are satisfied. The authors implemented their algorithm in the soot

framework [15] and evaluated against 17000 methods and all the types were successfully resolved. 99.8% of them were resolved in the first stage and only 0.2% of them required the second stage, and no method was required in the third stage.

Clem et al. [9] implemented symbolic code navigation for the GitHub platform which lets the user click on a named identifier in a source code and go to the definition of the entity. This code navigation is built upon tree-sitter, an incremental parsing tool that extracts name-binding information from source code. This approach indexed and stored name-binding information from each incremental push on the GitHub platform. This code navigation system supports nine programming languages. The system can process and index 1,000 pushes per minute and serve 30,000 requests of symbol lookup per minute.

Gasparini et al. [16] proposed an approach to enhance the GitHub interface for the pull request changeset under review. They introduced two lateral bar views to show the method definition and all other usages for any method invocation that is available in that changeset. To determine the appropriate method definition, they search the source files that are part of the changeset and they relied on *JavaSymbolSolver* [1] to resolve the binding information. *JavaSymbolSolver* is a part of the JavaParser project. JavaParser library parses Java source code and creates abstract syntax trees. Additionally, JavaSymbolSolver provides the functionality to connect each program element to its declaration.

2.2 Limitation of the existing approaches

In this section, we will discuss some of the key limitations of the existing literature and how our proposed approach will mitigate these limitations.

Consideration of external dependencies: The approaches proposed by Thummalapenta et al. [13] and Dagenais et al. [4] have attempted to resolve type information without considering external libraries and frameworks. However, projects typically depend on a large number of external libraries and frameworks. Leveraging the information collected from dependent external libraries and frameworks, we can identify binding information with more precision. Clem et al. [9] also consider only

[1] https://github.com/javaparser/javaparser

the internal method declarations of the project as candidates; hence, the approach cannot resolve method binding information, which is part of external dependencies as well as the Java API. On the other hand, our approach considers all the dependent external libraries and frameworks archivable, and the project archivable in order to identify the appropriate method-binding information.

Support of java version: PPA [4] supports Java 1.4, which is very old. Language features such as generics and auto-boxing are not available in Java 1.4. Hence, PPA does not provide support for such language features. Java has evolved drastically over the years. Java has introduced new language features (i.e., enumeration, variable arguments, functional interfaces, lambda expressions), and consideration of those features will help the precision of determining binding information. Our approach supports up to Java Version 11. Oracle provides long-term support for Java Version 11. After Java Version 11, currently, only Java Version 17 has long-term support.

Reliance on Inference Strategies: The approaches proposed by Thummalapenta et al. [13], Dagenais et al. [4], and Zhong et al. [12] relies heavily on limited number of different inference strategies. However, these inference strategies do not cover all language constructs. As a result, these approaches may suffer in accuracy for newer Java Versions. Instead, a systematic approach should be proposed which uses the grammar of the language to identify the binding information. Our approach relies on the Java language grammar (e.g., inheritance, consideration of the order of import statements) to solve the binding information.

Support for resolving formal type parameters for Generics: Generics is one of the highly utilized language features in Java. Resolving the appropriate types for formal parameters will help us understand the original types of arguments of a method or the return of the type of a method during invocation. JavaSymbolParser has very limited support for resolving formal type parameters declared on method definitions. JavaSymbolParser only considers type parameters that are directly passed during invocation. However, in newer language versions, type parameters are inferred from the type of the arguments or type of variable where the return type of the method invocation is assigned. Our approach provides extensive type inference support where we consider the parameterized invoker types, type of arguments, and type of assigned variable to determine the original type of formal type parameters.

Reliance on existing tools: The proposed approach GRAPA [12] depends on complete code

analysis tools. JavaSymbolParser is developed on top of JavaParser, an AST parsing library. On the other hand, the core philosophy of our approach is to build a tool that can extract data from different sources (e.g., Eclipse JDT Parser, JavaParser), construct our own representation, and perform the method binding information resolution. Our core underlying API is not tightly coupled with any parsing library. Hence, practitioners can integrate our API into their existing system with the minimum code change.

Chapter 3

Background

In this chapter, we will explain the key concepts which are related to our problem domain. We will also explain a few terminologies that we will use throughout the paper.

Abstract Syntax Tree(AST): Abstract Syntax Tree (AST) is a tree-based representation of the source code which preserves the structural and content-related information. Figure 3.1 illustrates an example of a java program and its corresponding Abstract Syntax Tree(AST). In this representation, each node, known as AST Node, represents a construct occurring in a source code. For java, there are a few well-known libraries (e.g., Eclipse JDT [1], JavaParser[2]) to generate AST from source code. Our approach provides functionality to take any method invocation AST node as input and extract all the necessary information that is required for our analysis and provide appropriate method binding information. Our current implementation supports AST nodes generated from Eclipse JDT as input. In this paper, we have followed the same name naming convention as Eclipse JDT for all AST nodes.

AST Node: AST Node represents source code constructs such as name, type, expression, statement, or declaration. From figure 3.1, we can see the AST representation where *CompilationUnit* is the root AST node, and *MethodInvocation* is the leaf AST Node. Our approach supports any method invocation expression as input in order to produce appropriate method-binding information for that expression. Table 3.1 shows the five types of method invocation expression AST Node and their representation on the source code.

[1] https://www.eclipse.org/jdt/overview.php
[2] https://javaparser.org/

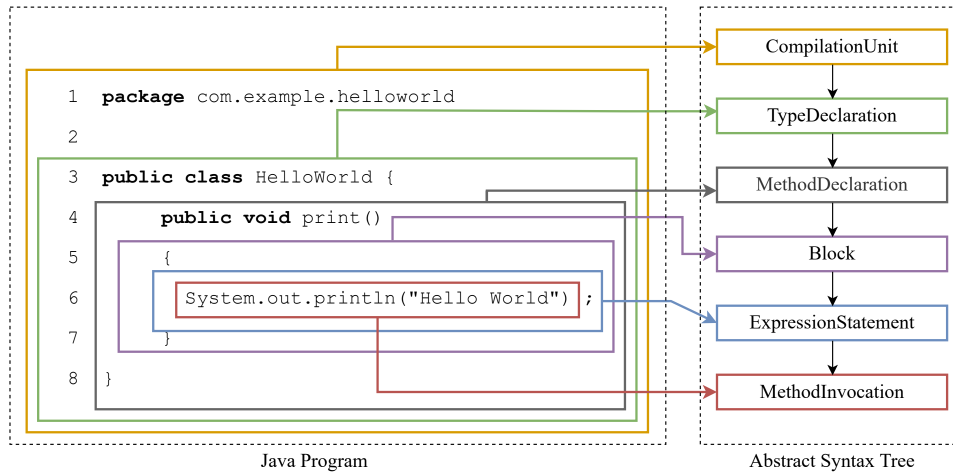


Figure 3.1: A Simple Java Program with the corresponding AST

Table 3.1: AST Node types for Method Invocation Expression

Type	Source Code Representation
MethodInvocation	Math.add(1, 2);
SuperMethodInvocation	super.print("hello-world");
ClassInstanceCreation	new Box();
ConstructorInvocation	this();
SuperConstructorInvocation	super();

Build System: Build system facilitates the compilation of code, executing tests, performing tasks, and producing deliverables. Typically build systems invoke commands in an orderly fashion in order to generate the deliverable. A deliverable can be an executable file or a packaged file format. Some build system also takes the responsibility of managing external dependencies such as third-party libraries the project is depending on. Typically, the Build system has a build specification file where users define the list of external dependencies as well as build-related configuration. The build system takes all the necessary configuration from the specification file and executes tasks such as compiling the projects, running tests, and finally generating the deliverable.

Maven: Maven[3] is a popular build system for java projects. Maven provides a unified and defined build process where it follows different steps or phases (e.g., validate, compile, test) to produce the deliverable, also known as a project artifact. The artifact can be a JAR or POM. Maven also provides support for managing external dependencies. The feature of resolving third-party libraries is facilitated by downloading the deliverable (e.g., JAR, POM) of external dependencies from specified remote artifact repositories. A POM-type deliverable consists of a list of jars. Maven uses a set of identifiers known as coordinates to uniquely identify a project and specify the packaging of the project deliverables. The set of identifiers are:

(1) **Group ID:** Group ID indicates the group or individual that created the project.

(2) **Artifact ID:** Artifact ID is the name of the project.

(3) **Version:** Version represents the project version.

In this paper, we will use artifact terminology to represent project maven coordinates. We have also used archivable and archive interchangeably to represent the packaged file format the project produces after a successful build.

Project-Object-Model (POM): Project-Object-Model or POM is a build specification file for Maven projects. It is an XML file that contains project and build configuration details. Typically POM file is located under the root directory. If the project is a multi-module project, then each sub-module will also contain a POM file. Usually, POM files are defined as *pom.xml* or files with *pom*

[3] https://maven.apache.org/

file extension. Figure 3.2 shows the structure of a typical POM file. From the figure, we can see the project artifact and all external project artifacts are defined under the dependency tag. Maven provides a plugin framework to execute every phase of the build life-cycle. In the figure, a *maven-compiler-plugin* is defined to specify the java language version for the compilation as well as for the build.

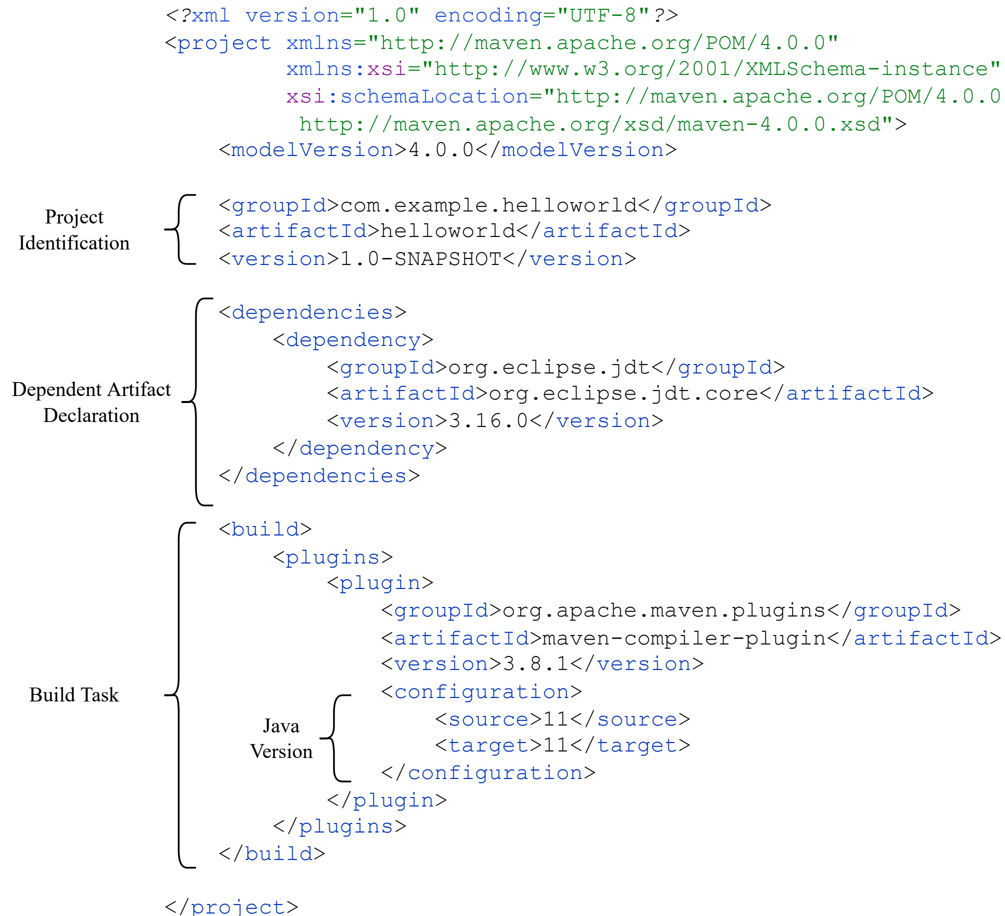


Figure 3.2: POM File

Super POM: Maven has a default build configuration file, which is known as Super POM. All the user-defined POM files will extend the build configuration from Super POM. So user defined POM files can override the default build configuration.

Effective POM: Effective POM is the culmination of Super POM and user-defined POM. Effective POM is the final output where all the default configurations will remain unless are overridden

by the user-defined POM.

Maven Artifact Repository: Maven Artifact Repository can hold the deliverable artifacts (e.g., JAR, WAR) produced by the build system. Maven offers two types of artifact repositories.

(1) **Local:** Local artifact repository is a directory for storing the artifacts, created on the user's system during Maven installation. This directory can contain artifacts downloaded from remote artifact repositories and temporary build artifacts which are not been released yet.

(2) **Remote:** Remote artifact repository holds the published project artifacts which can be accessed using various protocols (e.g., HTTP). Practitioners can host private or public repositories to host their artifacts. Maven offers one of the largest public artifact repository servers (i.e., *repo.maven.apache.org*).

Gradle: Gradle[4] is another popular build automation tool for java projects. The build process is performed via the execution of different tasks (e.g., compile, test, build). A task can have multiple dependent tasks. Gradle will ensure the execution of dependent tasks before executing the task. Every task will be executed once. Gradle allows project identification similar to Maven. Maven coordinates are used to define project as well as find external third-party dependencies from remote artifact repositories. Gradle build specification files can be written in groovy DSL[5] or Kotlin. Typically build specification file will be named as *build.gradle* or *build.gradle.kts*. Gradle also supports multi-module projects where each module will also contain a build specification file.

Figure 3.3 shows the structure of a typical Gradle build file written in groovy DSL. Leveraging the *command-chain* feature of groovy all the configurations are written. Using this feature any method invocation with an argument can be written as space-separated. For example *group('org.example')* can be written as *group 'org.example'*. From the figure, we can see the dependent artifacts declaration as well as the Java version configuration for compilation and build.

Gradle Tooling API: Gradle provides a programmatic API, known as Tooling API for executing Gradle tasks programmatically. This API is used by the majority of modern Integrated Development Environments (IDE) to integrate Gradle build process into the development environment. This

[4] https://gradle.org/

[5] http://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html

Figure 3.3: Gradle Build File

tooling API provides more granular control over command line invocation. This tooling API can also run independently of the version of the Gradle project.

Method Binding Information: Method binding refers to identifying a method or constructor declaration from its invocation reference. Method binding offers information about the declared form of the method (e.g., name of the class the method belongs to, type of the arguments of the method, return type of the method). Additionally, Method binding contains all the resolved type parameters of generic methods. The objective of our approach is to provide method binding as a data transfer object (DTO) that will contain all the information extracted from the method declaration. We capture all method binding information decompiling class files from java core API packages, project deliverables, and external project dependency deliverables.

Invocation Context: In this paper, invocation context refers to the context where the method invocation occurred. We collect information from the invocation context such as type of arguments, type of method invoker, type arguments, and invocation context class, which is the class where the method was invoked. Additionally, we also collect all the superclasses and inner classes of the

invocation context class.

Variadic Argument: Java programming language supports the functionality of passing a variable number of arguments to a method. Java will consider all the arguments as an array argument. This type of argument is called a variadic argument. In Java, variadic argument is also known as varargs argument. In this paper, we have used variadic argument and varargs argument interchangeably.

Chapter 4

Approach

This chapter will outline the approach used to resolve the appropriate method-binding information. There are two stages to our approach. We can view each step as an API call where the user provides input to receive a set of outputs. The first stage 4.1, encompasses the extraction of the project's Java version and dependent external library artifacts from the build system and the storing process of all class metadata (e.g., fields, methods, superclass hierarchy). We collect class metadata by extracting Java core packages based on the project's Java version and archives of all dependent artifacts. In the second stage, 4.2, we perform the extraction of accurate method-binding information for method invocation expression. The approach's overview is shown in figure 4.1.

4.1 Artifact & Java Version Extraction

To extract external artifacts and the Java version of the project, as a first step, we determine the project's build system. We take different approaches based on the project's build system to extract the Java version and dependent artifacts. Our methodology currently supports the project's Java version and artifact extraction from the Maven and Gradle build systems. After the extraction process, we retrieve all the archive files of dependent artifacts from Maven's public artifact repositories. We extract all the class metadata from archive files using a byte-code decompiler and store the metadata in our system. Figure 4.2 illustrates an overview of this procedure.

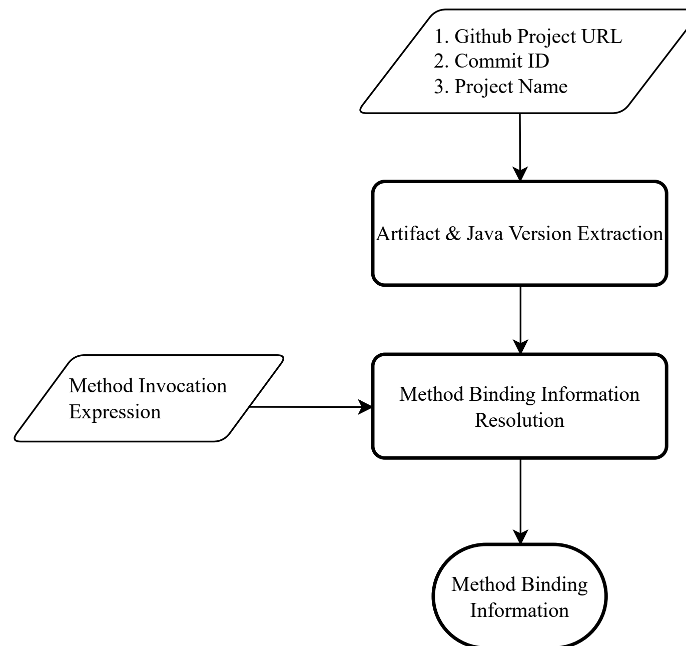


Figure 4.1: Overview of our approach

4.1.1 Input

Our approach will require three inputs for the extraction of the project's java version and dependent artifacts. The three inputs are:

(1) GitHub Project URL: The URL that identifies the project on GitHub.

(2) Commit ID: specifies the specific commit ID in the commit history.

(3) Project Name: identifies the project in our system.

4.1.2 Determination of Build System

There are a few distinguishing characteristics that can be used to identify the build system of a project. The presence of a *pom.xml* file or a file with the *.pom* extension in the root directory of the project will indicate a Maven build system. On the other hand, in a Gradle build system, the root directory will have a *build.gradle* or *build.gradle.kts* file.

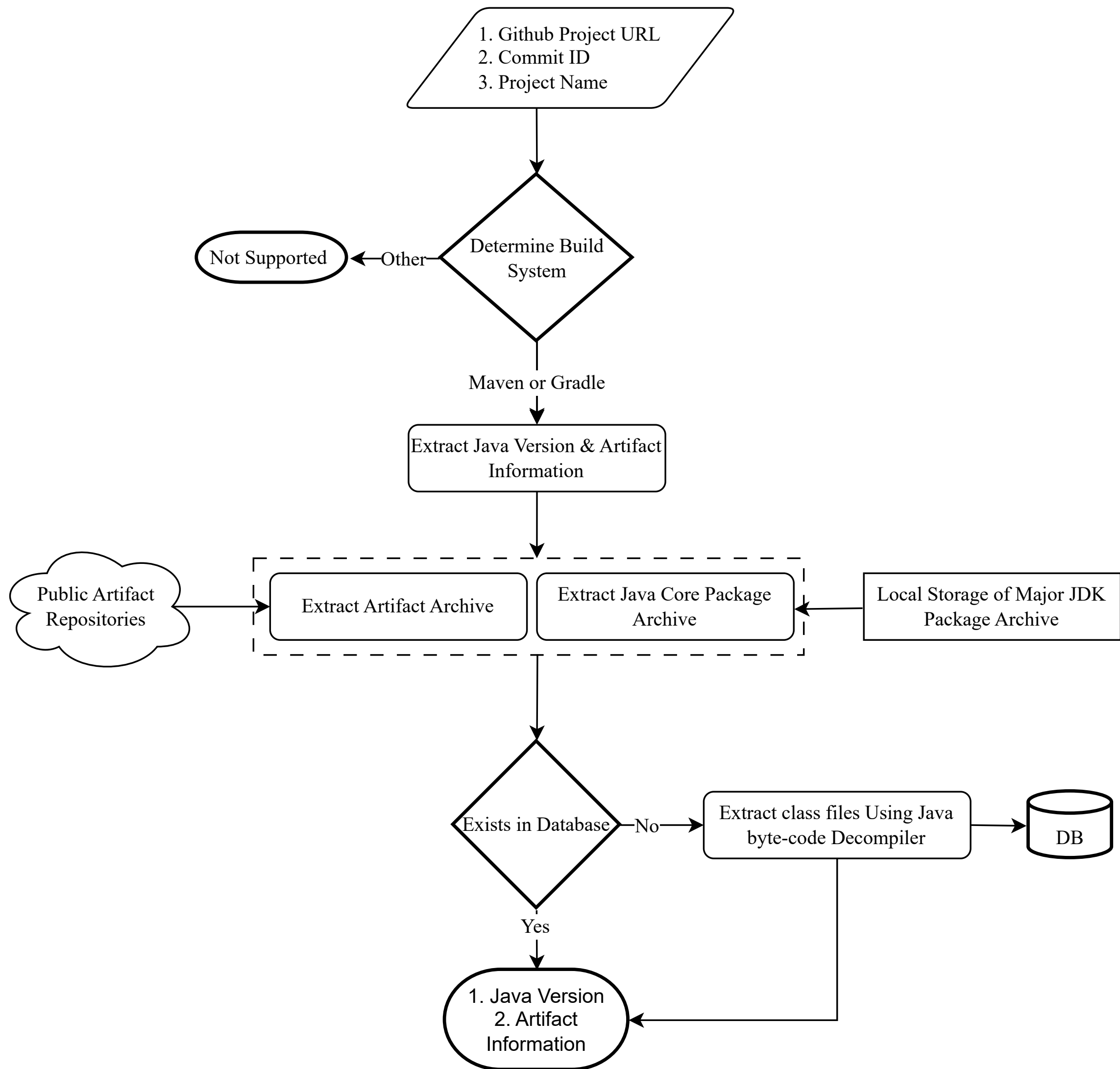


Figure 4.2: Overview of Java Version & Artifact Extraction

4.1.3 Extraction of Java Version & Artifacts

Depending on the type of build system, the procedure for extracting the Java version and dependent artifacts from build specifications varies. We additionally consider the project as another dependent artifact, as we wish to consider all the packages of the project to determine the appropriate method binding.

For Maven build systems, we need to parse POM files to determine the project's Java version and all external dependencies. The Gradle Tooling API can be used to retrieve the Java version and project dependencies from any Gradle build system.

Maven Build System To extract artifacts and the Java version of the project, we depend on Effective POM. Effective POM is a culmination of Super POM and user-defined POM configuration. In order to generate Effective POM, we need to extract all the POM files from the project. For the extraction of build configuration files (i.e., POM, build.gradle) we support two functionality.

(1) **Remote Fetch:** Remote fetch functionality allows us to use GitHub API to fetch only the build configuration files from GitHub repositories.

(2) **Local Fetch:** Local fetch functionality allows us to clone the project directory locally and perform the extraction of build files from the locally cloned directory.

We have implemented two API endpoints to provide the support for selecting two different fetch types 5.1.

After the extraction of all the POM files, we perform the generation of Effective POM using Apache Maven Invoker [1]. Apache Maven Invoker provides an API to perform the tasks by building up a conventional Maven command line from the options given. Finally, we parse the XML file to extract the Java version and dependent artifacts information.

Gradle Build System For Gradle build systems, we depend on Gradle tooling API to extract the Java version of the project and dependent artifacts information. We also support two types of fetch mechanisms for Gradle build systems. For remote fetch type extraction of build files, in certain

[1]https://maven.apache.org/shared/maven-invoker/

scenarios, extracting additional custom build-related files is required. The build scripts usually contain the location of these additional files. We provide a simple regex-based operation to perform the extraction of all build-related files.

4.1.4 Extraction Of Java Core Packages

Every Java Development Kit (JDK) includes core API packages (e.g., *java.lang*, *java.util*) that can be accessed in projects without declaring any external dependencies. We maintain local storage of all the archives of all major Java versions. For all the Java versions up to 8, core packages are archived as *jar*. For JDK 9 and later versions, a new archive format called *jmod* is introduced, which can contain additional files in addition to classes and resource files. We collect the Java core package archives based on the project's Java version.

4.1.5 Extraction of Artifact Archives

We leverage the support of the Apache Maven Artifact Resolver [2] to extract the artifact archives from public artifact repositories. We can package a project's deliverables in different types (e.g., JAR, POM, maven-plugin, bundle). Apache Maven Artifact Resolver helps us in the extraction of all the related archives for the project. For example, if a project is packaged as POM file, there can be sub-module archives that need to be extracted as well. Apache Maven Artifact Resolver helps us in the extraction of those dependent artifact archives.

4.1.6 Storing process

We extract the class metadata from all the archives of the Java core packages and dependent artifacts. We perform a check to verify whether our storage system already contains the archive's class metadata, and if we cannot find any evidence of its existence in our storage system, we continue the process of extracting class metadata from the archive. We utilize a byte-code decompiler to extract all the class metadata and store the data in our system.

[2] https://maven.apache.org/resolver/index.html

4.1.7 Output

The extracted information, such as the project's Java version and dependent artifact information, is sent as output. The next stage will require this extracted information as inputs to perform subsequent operations.

4.2 Method Binding Information Resolution

In this section, we explain the process of identifying appropriate method-binding information from any method invocation expression. Figure 4.3 depicts the procedure we followed to generate the correct method binding information. The entire process can be divided into three stages.

(1) Invocation context information extraction

(2) Identify appropriate method declaration and generate method binding information

(3) Post-processing of method binding information

4.2.1 Invocation context information extraction

In the first stage, we extract information from the invocation context that will help us identify the appropriate method declaration candidate. We will present the invocation context-specific information we collect and explain the necessity for locating the appropriate candidate.

Import Statement Extraction We extract all the import statements of the source file. Import statements are used to bring a class, a method, or classes inside a package into visibility for a class file. Visibility is required to declare a class or method inside a class file without a fully qualified name. Hence, extracting import statements will help us identify all the possible class candidates to which the declared method may belong.

Invocation Context Class Information Extraction An invoked method can be declared on the invocation context class or any of its superclasses or inner classes. In such scenarios, import statements are not required for a class's visibility. Thus, we also collect four key invocation context class information.

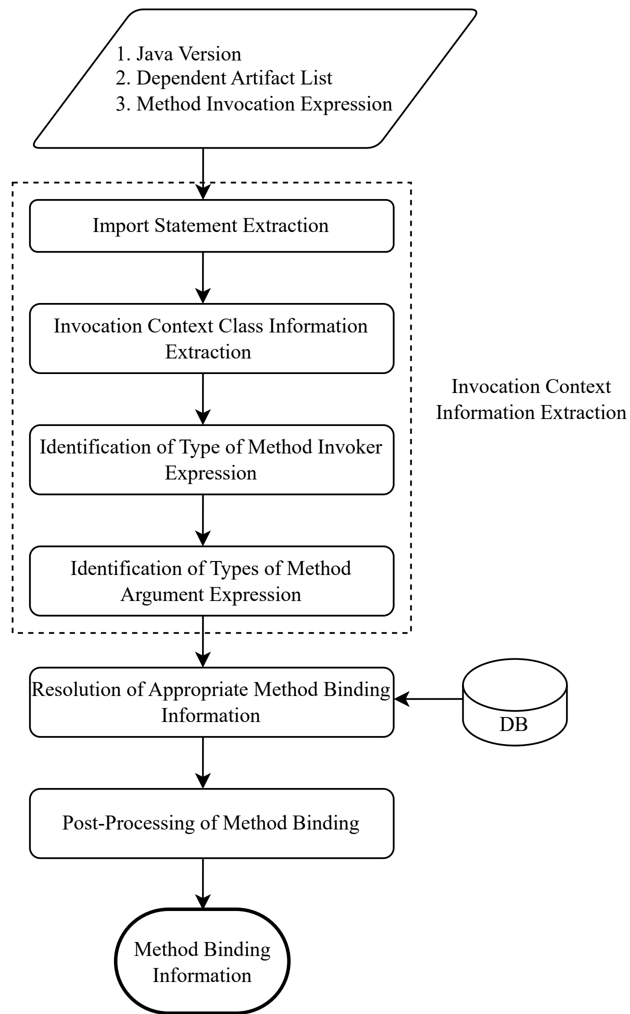


Figure 4.3: Overview of Method Binding Information Resolution

(1) *Qualified Name of Invocation Context Class:* The invocation context class is the class where the method invocation happened. If the invocation resides in a nested inner class, the immediate enclosing inner class will be the invocation context class.

(2) *Qualified Class Name Hierarchy:* Java is an object-oriented programming language and supports inheritance. Thus, a child class can have access to any attributes or methods of superclasses, and we have to consider the superclass hierarchy of the invocation context class in order to determine the appropriate method declaration. We collect all the superclass names and their inner class names, preserving their hierarchical order.

(3) *Invocation Context Class Declaration Order:* We also collect an ordered list of qualified names of all classes in the superclass hierarchy, preserving their declaration order in the class file. In the case of multiple candidates where all other conditions are satisfied, the class with the earlier position in the declaration order will have priority.

(4) *Invocation Context Formal Type Parameter List:* We also extract a list of *FormalTypeParameterInfo* for a generic invocation context class. We use the list of *FormalTypeParameterInfo* to resolve the formal type parameters in the post-processing of method binding information 4.2.3 stage.

Identification of Type of Method Invoker Expression A method invocation can have an invoker expression. The invoker expression represents the expression upon which the method invocation occurred. An invoker expression can be a declared variable, a fully qualified class name, or a field instance. In figure 3.1, *System.out* represents the invoker expression. In this instance, *out* is the name of a field instance declared on *java.lang.System* class, and the qualified class name of the type of the field instance is *java.io.PrintStream*. Identifying the appropriate type representation of the invoker expression will help us narrow down the eligible classes to which the method declaration belongs.

Identification of Types of Method Argument Expressions An invocation of a method may require one or more arguments. Identifying the type representation of the method argument expression

will help us identify the appropriate candidate. We compare the type of argument expression with the type of declared method parameters to identify the appropriate method declaration. Listing 1 demonstrates a case in which we can identify the appropriate method declaration by comparing types of arguments among multiple overloaded method declarations.

Listing 1 Identification of accurate method declaration from Type of Arguments in Invocation Context

```
//Method Invocation Context
public strictfp class Range implements Serializable {
...
    private static double min(double d1, double d2) {
        ...
        return Math.min(d1, d2);
    }
}
// Method Declaration Context
public final class Math {
...
    public static double min(double a, double b) {
        ...
    }

    public static float min(float a, float b) {
        ...
    }
}
```

Type Representation In order to represent the type of any expression, we have developed our own type representation system, which allows us to keep the necessary data for our computation. We have converted the types of invoker expressions as well as the types of method arguments into our type representation. We have used this uniform and simplified representation of the type system throughout our entire approach. Table 4.1 shows the complete list of type representations in our system. We have introduced a total of ten representations to depict all the Java types.

PrimitiveTypeInfo represents the primitive types (e.g., int, char, boolean), and *QualifiedTypeInfo* represents a fully qualified class name. In addition, we also introduced *SimpleTypeInfo*, which will contain the non-qualified name of the class. In Java, any generic class instantiated with a type

Table 4.1: Type Representation

Type Name	Represents
PrimitiveTypeInfo	All primitive types (e.g., int, char, boolean)
QualifiedTypeInfo	Any object data types
SimpleTypeInfo	Object data types used for user input
ParameterizedTypeInfo	Any parameterized type with type arguments
FormalTypeParameterInfo	Generics formal parameters
ArrayTypeInfo	array
VarargTypeInfo	varargs
NullTypeInfo	null
FunctionTypeInfo	lambda expression, Method References
VoidTypeInfo	void

argument is called parameterized type. To represent a parameterized type, we have used *ParameterizedTypeInfo*. In *ParameterizedTypeInfo* representation, we store the fully qualified name of the generic class together with a list of type arguments. Each type argument can itself be a type representation (i.e., QualifiedTypeInfo, FormalTypeParameterInfo, ParameterizedTypeInfo, or ArrayTypeInfo). *FormalTypeParameterInfo* represents the formal type parameter and we store the name of the type parameter and base type info in this representation. In Java, we can assign an upper bound (i.e., extends) or lower bound (i.e., super) class in the definition of a formal type parameter. In our representation, the bound class name is stored as the base type. The default base type is *java.lang. Object*. *ArrayTypeInfo* and *VarargTypeInfo* represent array and variadic argument respectively and we store the class name of type inside our representation. Additionally, *ArrayTypeInfo* will contain the dimension of the array. We represent any lambda expression or method references (e.g., constructor references, static method reference) with *FunctionTypeInfo* which only stores the return type and argument types of the respective methods. Finally, *NullTypeInfo* and *VoidTypeInfo* are used to represent the null and void type, respectively.

To convert an expression into our type representation, we need to identify the appropriate binding information for the expression. From the binding information, we can construct our type representation. There are two types of binding information we may need to extract from an expression.

(1) Identifying Class Binding Information from type or expression name.

(2) Identifying Field Binding Information from expression name.

Identifying Class Binding Information from Type or Expression Name Identifying class-binding information from type or expression names is a multi-stage process. The visibility of a class or package in a partial program requires one of these conditions.

(1) The class or package is internal to the project.

(2) The class or package is internal to the Java core API.

(3) The class or package is declared in dependent artifact archives.

In the Artifact and Java Version Extraction stage 4.1, we identify the project's Java version, dependent artifact information, and the project's artifact information. Based on the above-mentioned conditions, we can claim that the class of any defined expression in the partial program will belong to packages that are declared in the Java core API archive, dependent artifact archive, or project archive.

In addition, there are two ways a type or class can be declared in an expression (i.e., simple type and qualified type). In qualified type representation, the fully qualified class name is available in the expression. On the other hand, in simple type representation, only the class name is available in the expression. However, the visibility of the class name requires one of the following conditions:

(1) The fully qualified class name is declared on the import statement.

(2) The class is declared as one of the inner classes inside the invocation context class.

(3) The class is declared in one of the parent classes of the invocation context class.

Based on the conditions mentioned above, we have formulated our algorithm. Algorithm 1 shows the algorithm for detecting class binding information. Our algorithm requires dependent artifact information, Java version, extracted import statements 4.2.1, invocation context class information 4.2.1, and type or expression name as input to extract appropriate class binding information. We expect the output produced in the section 4.1 should be used as inputs for dependent artifact information and the Java version.

On line 1, we initialize an empty eligible class binding information list. On line 2, we extract all the class binding information of the Java core API archives and dependent artifact archives from

our storage. If the expression is a qualified type where the expression contains the fully qualified class name, we perform a modification on line 6 where we add the fully qualified class name in $importedClassNameList$ and convert the qualified name of the expression to the simple name. A simple name only contains the name of the class without package information. In lines 3 and 4, we extract the imported class names and package names from the import statement. In line 8, we iterate over the class binding list that we constructed on line 2 and check the existence of the qualified name of particular binding information in the list of imported class names, packages, or invocation context class hierarchy on line 10. Upon fulfillment of any condition, we perform the name matching on line 7 and add it to our eligible class binding list. Finally, we perform priority-based filtration on the eligible class binding list to narrow down the eligibility and identify the appropriate class binding information.

Algorithm 1 Identify class binding information c from type or expression name n

Input: D is a set of dependent artifact information, J is Java Version, $invocationConCI$ is a Data-Object-Model containing invocation context class information, I is a list of import statements, n is the name of the type or expression

Output: c is the eligible class from type name N

1: Initialize $eligibleClassBindingInfoList$
2: $classBindingInfoList \leftarrow getAllClassBindingInfo(D, J)$
3: $importedClassNameList \leftarrow getAllImportedClass(I)$
4: $importedPackageList \leftarrow getAllImportedPackage(I)$

5: **if** $isQualifiedType(n)$ **then**
6: $\quad n \leftarrow processModification(n, importedClassNameList)$
7: **end if**

8: **for** $i = 1, i <= classBindingInfoList.size(), i++$ **do**
9: $\quad cBindingInfo = classBindingInfoList.get(i)$
10: $\quad$ **if** $importedClassList.contains(cBindingInfo.getQName())$
or $importedPackageList.contains(cBindingInfo.getPackageName())$
or $invocationConCI.getClassNameHierarchy().contains(cBindingInfo.getQName())$ **then**
11: $\quad\quad$ **if** $cBindingInfo.getName().equals(n)$ **then**
12: $\quad\quad\quad eligibleClassBindingInfoList.add(class)$
13: $\quad\quad$ **end if**
14: $\quad$ **end if**
15: **end for**

16: $eligibleClassBindingInfoList \leftarrow priorityBasedFiltering(eligibleClassBindingInfoList)$
17: $c \leftarrow eligibleClassBindingInfoList.get(0)$

18: **return** c

For priority-based filtering, we have followed a heuristics-based approach adopted from the Java language convention. We processed the filtering in four steps in an orderly fashion and try to narrow down the eligible class list.

Algorithm 2 describes the priority-based filtration process. On line 2, we check whether any eligible class binding information is declared on the import statement as a class. In that scenario, we prioritize that class binding information as eligible and return it as output. On line 5, we check whether the eligible class binding information is declared in the invocation context class hierarchy 4.2.1. If we can find any eligible candidate, we prioritized those class binding information as eligible and return it as output. In the next step, on line 8, we check if any class binding information is declared under the Java core API packages (i.e., `java.lang`). In such a scenario, we consider those class binding information as eligible and return it as result. Finally, on line 11, we check whether the type or expression itself is a qualified type. In that scenario, we prioritize the eligible class list using the matched fully-qualified name and return it as result.

Algorithm 2 Priority Based Filtering on $cBindingInfoList$

1: **function** PRIORITYBASEDFILTERING($importedClassNameList$,$invocationConCI$, n, $cBindingInfoList$)

2: **if** Any $getClassQNameList(cBindingInfoList)$ exists in $importedClassNameList$ **then**
3: **return** $getFilteredListOnlyDeclaredInImports(cBindingInfoList, importedClassNameList)$
4: **end if**

5: **if** Any $getClassQNameList(cBindingInfoList)$ exists in $invocationConCI.getClassQNameHierarchy()$ **then**
6: **return** $getFilteredListOnlyPartOfClassHierarchy(cBindingInfoList, invocationConCI.getClassQNameHierarchy())$
7: **end if**

8: **if** $cBindingInfoList$ exists in the Java core API package **then**
9: **return** $getFilteredListOnlyPartOfJavaDefaultPackage(cBindingInfoList)$
10: **end if**

11: **if** $isQualifiedType(n)$ **then**
12: **return** $getFilteredListMatchesQualifiedTypeName(cBindingInfoList, n)$
13: **end if**

14: **return** $cBindingInfoList$
15: **end function**

Identifying Field Binding Information from Expression We perform the extraction of field binding information in five stages. We can construct the type representation from field binding information. Hence, identifying appropriate field binding information will help us construct an appropriate type representation of the field instance. We identify appropriate binding information from different scopes or perspectives. We gradually increase the scope to resolve appropriate binding information. If we can find any eligible candidate in any of the scopes, we return the candidates as output; otherwise, in the end, we return an empty list.

Algorithm 3 shows the five-stage approach of identifying appropriate field binding information from expression. In lines 1-3, we initialize $fBindingList$ and extract the class names and package names from import statements. On line 4, we extract the field name from the expression. The expression can contain a simple or fully qualified class name associated with the field name. Therefore, if a fully qualified class name is available we extract the class name and append it to $importedClassList$ inside the $processFieldName$ method.

We start our process through the initialization of $fBindingList$ which will contain the appropriate field binding information list. On lines 2 and 3, we perform the extraction of qualified class names and package names defined in import statements. In the first stage, we consider the field expression scope. A field instance may contain a qualified class name associated with it in the expression. In such a scenario, we start our search process for finding appropriate field binding information from the fully qualified class name. On line 8, we invoke the function $getFieldBindingInfoListForExpressionScope$ to get the eligible field binding candidates. Algorithm 4 presents the search process. We start our search for the field instance from the qualified class name. If we can find a field instance with $fieldName$ in that class, we will consider the instance an eligible candidate and return from the function. Otherwise, we in an iterative fashion, traverse the superclasses of the qualified class name and try to find the field instance. If we cannot find any instance that matches $fieldName$ we return an empty list.

At the end of stage 1, if we cannot find any eligible field binding information we move to the next stage. The scope of this stage is the method invocation context where we assume the field instance is declared in the method invocation context. On line 15, We start our search for the field from the invocation context class and traverse all superclasses. If we can identify eligible field

Algorithm 3 Identify field binding information f from expression e

Input: D is a set of dependent artifact information, J is Java Version, $invocationConCI$ is a Data-Object-Model containing instance context class information, I is a list of import statements, e is the field expression

Output: $fBindingList$ is the list of eligible field binding information that can be inferred from the expression e

1: Initialize $fBindingList$
2: $importedClassList \leftarrow getAllImportedClass(I)$
3: $importedPackageList \leftarrow getAllImportedPackage(I)$
4: $fieldName \leftarrow processFieldName(e, importedClassList)$
5: ▷ Stage 1: Field expression scope
6: **if** expression e contains a fully qualified class name with field name **then**
7: $qClassName \leftarrow extractQualifiedClassName(e)$
8: $fBindingList \leftarrow getFieldBindingInfoListForExpressionScope(D, J, qClassName, fieldName)$
9: **if** $isNotEmpty(fBindingList)$ **then**
10: **return** $fBindingList$
11: **end if**
12: **end if**

13: ▷ Stage 2: Method Invocation context scope
14: **if** $isNotEmpty(invocationConCI.getClassNameHierarchy())$ **then**
15: **while** Iterate over $invocationConCI.getClassNameHierarchy()$ **do**
16: $classSet \leftarrow instanceConCI.getClassNameHierarchy().get(index)$
17: $fBindingList \leftarrow getFieldBindingInfoList(D, J, classSet, fieldName)$
18: **end while**
19: **if** $isNotEmpty(fBindingList)$ **then**
20: **return** $fBindingList$
21: **end if**
22: **end if**

23: ▷ Stage 3: Directly imported class scope
24: $fBindingList \leftarrow getFieldBindingInfoList(D, J, importedClassList, fieldName)$
25: **if** $isNotEmpty(fBindingList)$ **then**
26: **return** $fBindingList$
27: **end if**

28: ▷ Stage 4: Imported package scope
29: $fBindingList \leftarrow getFieldBindingInfoList(D, J, importedPackageList, fieldName)$
30: **if** $isNotEmpty(fBindingList)$ **then**
31: **return** $fBindingList$
32: **end if**

33: ▷ Stage 5: Super classes of all imported classes scope
34: $classSet \leftarrow importedClassList$
35: **while** $classSet$ has superclass or interface AND $isEmpty(eligibleFieldList)$ **do**
36: $classSet \leftarrow getImmediateSuperClassOrInterface(classSet)$
37: $fBindingList \leftarrow getFieldBindingInfoList(D, J, classSet, fieldName)$
38: **end while**

39: **return** $fBindingList$

binding information, we return from the algorithm. Otherwise, we move to the next stage.

In stage 3, on line 24, we will only consider the directly imported classes as a scope and try to find the field binding information. On line 29, we move to stage 4, we look into all the classes of imported packages and try to find the field instance, and if there is any eligible field binding information we will return from the algorithm.

Finally, in stage 5, we will look into all the superclasses of all the classes (both directly imported or via package import). We will gradually traverse into the superclass hierarchy until we reach `java.lang.Object`. If we can find field binding information, we can return it as an eligible candidate otherwise we will return an empty list.

Algorithm 4 Function for finding field binding information from Field Expression Scope

1: **function** GETFIELDBINDINGINFOLISTFOREXPRESSIONSCOPE(D,J,$qClassName$,$fieldName$)
2: initialize $eligibleFieldBindingInfoList$, $classSet$
3: $classSet.add(qClassName)$

4: **while** $isNotEmpty(classSet)$ AND $isEmpty(eligibleFieldBindingInfoList)$ **do**
5: $eligibleFieldBindingInfoList \leftarrow getFieldBindingInfoList(D, J, classSet, fieldName)$

6: **if** $isEmpty(eligibleFieldBindingInfoList)$ **then**
7: $classSet \leftarrow getSuperClassSet(classSet)$
8: **end if**
9: **end while**

10: **return** $eligibleFieldBindingInfoList$
11: **end function**

4.2.2 Resolution of Appropriate Method Binding Information

After acquiring all invocation context information, we will perform the resolution of appropriate binding information. We have introduced a six-stage approach to identify the appropriate

method-binding information similar to the field binding resolution. Algorithm 5 shows the complete approach. We will explain each stage separately with the corresponding algorithm of each stage.

In lines 1-4, we initialize our list, extract the classes and packages from import statements, and process the method name. From line 6, we check the existence of the method invoker. If the method invocation has an invoker expression and we can identify the type of the invoker (4.2.1), we will try to look for an eligible method binding from the invoker expression scope.

Stage 1: Invoker expression scope Algorithm 6 shows the first stage where we try to resolve the method binding information from the type of invoker expression. At this stage, after initialization, we start from the invoker class and then we iterate through the superclass hierarchy of the invocation context class. We traverse all parent classes or interfaces of the invocation context class, followed by traversing their parents. We continue this process until we complete the superclass hierarchy. We search for method binding information based on the method name and the number of arguments on line 5.

If the type class name of the invoker expression is invocation context class, we need to set $true$ to the attribute $invocationContextAttribute$ for all eligible method binding information [line 6]. We utilize this attribute in the filtration process (4.2.2). On lines 9 and 10, we perform post-modification to the eligible method binding information and apply the filtration process in order to identify the appropriate method binding. We have explained the post-modification (4.2.2) and filtration (4.2.2) processes in detail in later sections.

In this iterative process of traversal of the superclass hierarchy, we may find eligible method candidates that partially match all the selection criteria. However, there can be other eligible method binding information candidates in the hierarchy that are more accurate. We can define such eligible method bindings as deferred method bindings. For deferred method bindings, we have certain conditions. We have explained the deferred conditions in a later section. We try to hold deferred method bindings until we complete the traversal. If we still can't find any non-deferred eligible method binding candidates, we will find the appropriate method binding from the deferred set. On line 11, we performed the check for the criteria for being deferred and stored the candidates in a

Algorithm 5 Identify method binding $mBinding$ from method name expression $mNameExp$

Input: D is a set of dependent artifact information, J is Java Version, I is a list of import statements, $mNameExp$ is the method name expression, $numberOfArgs$ is the number of the method arguments, $invokerType$ is the type of method invoker, $argTypeList$ is the list of argument types of the method, $invocationConCI$ is an object containing all invocation context class information

Output: $eligibleMethodList$ is the list of eligible methods that can be inferred from method name $methodName$

1: Initialize $mBindingList$
2: $importedClassList \leftarrow getAllImportedClass(I)$
3: $importedPackageList \leftarrow getAllImportedPackage(I)$
4: $mName \leftarrow extractMethodName(mNameExp)$
5: ▷ Stage 1: Method invoker expression scope
6: **if** $invokerType$ exists **then**
7: $eligibleMethodList \leftarrow getMBindingListforInvokerExpressionScope(mName, numberOfArgs, invokerType, argTypeList, invocationConCI.getClassNameHierarchy())$
8: **if** $isNotEmpty(eligibleMethodList)$ **then**
9: **return** $eligibleMethodList$
10: **end if**
11: **end if**
12: ▷ Stage 2: Method invocation context scope
13: **if** $invocationContextClassHierarchy$ is not empty **then**
14: $eligibleMethodList \leftarrow getMethodListForInvocationContextScope(mName, numberOfArgs, invokerType, argTypeList, invocationConCI.getClassNameHierarchy())$
15: **if** $isNotEmpty(eligibleMethodList)$ **then**
16: **return** $eligibleMethodList$
17: **end if**
18: **end if**
19: Initialize $deferredMethodSet$
20: ▷ Stage 3: Directly imported class scope
21: $eligibleMethodList \leftarrow getMethodListFromImportedClasses(mName, numberOfArgs, invokerType, argTypeList, deferredMethodSet, importedClassList)$

22: **if** $isNotEmpty(eligibleMethodList)$ **then**
23: **return** $eligibleMethodList$
24: **end if**
25: ▷ Stage 4: Inner classes of directly imported class scope
26: $eligibleMethodList \leftarrow getMethodListFromInnerClass(mName, numberOfArgs, invokerType, argTypeList, deferredMethodSet, importedClassList)$

27: **if** $isNotEmpty(eligibleMethodList)$ **then**
28: **return** $eligibleMethodList$
29: **end if**

30: ▷ Stage 5: Imported package scope
31: $eligibleMethodList \leftarrow getMethodListForPackageScope(mName, numberOfArgs, invokerType, argTypeList, deferredMethodSet, importedPackageList)$

32: **if** $isNotEmpty(eligibleMethodList)$ **then**
33: **return** $eligibleMethodList$
34: **end if**
35: ▷ Stage 6: Super classes of all imported classes scope
36: $eligibleMethodList \leftarrow getMethodListForSuperClassScope(mName, numberOfArgs, invokerType, argTypeList, deferredMethodSet, importedPackageList, importedClassList)$

37: **return** $eligibleMethodList$

separate list for further processing.

After the filtration and deferral processes, if we still have any eligible candidates, we break the iteration process to return the result. Otherwise, we try to look into parent classes and interfaces.

After traversal of the superclass hierarchy, if only the set of deferred method binding information is available, we perform a prioritization over the deferred method binding set and return the result.

Stage 2: Invocation Context Scope The second stage is the invocation context scope. Our main focus in this stage is on the method invocation context class hierarchy. On line 13 of the algorithm 5 we check the availability of invocation context information and proceed to identify eligible method bindings in that scope.

Algorithm 7 shows the process of identifying eligible method bindings from the invocation context class hierarchy. We iterate over the invocation context class hierarchy that was created in the previous section 4.2.1. We start with the invocation context class and all of its inner classes. On line 4, we try to fetch all eligible method bindings from those classes based on the method name and the number of method arguments.

We also $true$ to the attribute $invocationContextAttribute$ for all eligible method bindings if they are declared on the invocation context class. On lines 7 and 8, we perform the same post-modification and filtration process as in the previous stage. We also perform a deferred criteria check for eligible method bindings (line 9). However, we allow deferring eligible method bindings for any parent class or interface in this stage, as any eligible method binding, even though it satisfies deferred criteria, will have priority. For deferred method bindings, we store them in a separate list

Algorithm 6 Algorithm for finding method binding list from invoker expresssion

Input: $mName$ is the name of the method, $numberOfArgs$ is the size of the method arguments, $invokerType$ is the type of method invoker, $argTypeList$ is the list of argument types of the method, $invocationContextClassHierarchy$ is an ordered set of invocation context class hierarchies,

Output: $eligibleMethodList$ is the list of eligible method binding list

1: initialize $classSet$, $deferredMethodSet$, $classNameDeclarationOrderList$, $eligibleMethodList$
2: $classSet.add(invokerType.getClassName())$
3: $classNameDeclarationOrderList.add(classSet)$

4: **while** $isNotEmpty(classSet)$ AND $isEmpty(eligibleMethodList)$ **do**
5: $eligibleMethodList \leftarrow getQualifiedMethodList(D, J, classSet, mName, numberOfArgs)$

6: **if** check $invokerTypeInfo$ exists in $invocationContextClassHierarchy.get(0)$ **then**
7: $setInvocationContextClassAttribute(eligibleMethodList)$
8: **end if**

9: $postModification(eligibleMethodList)$
10: $eligibleMethodList \leftarrow filter(eligibleMethodList, invokerType, argTypeList)$

11: **if** $hasAllDeferredCriteria(eligibleMethodList)$ **then**
12: $deferredMethodSet < -eligibleMethodList$
13: $eligibleMethodList.clear()$
14: **end if**

15: **if** $isEmpty(eligibleMethodList)$ **then**
16: $superClassSet \leftarrow getSuperClassSet(classSet)$
17: $processSuperClassClassOrder(superClassSet, classSet)$
18: $classSet \leftarrow superClassSet$
19: **end if**
20: **end while**

21: **if** $isEmpty(eligibleMethodList)$ AND $isNotEmpty(deferredMethodSet)$ **then**
22: $prioritizeDeferredMethods(deferredMethodSet, classNameDeclarationOrderList)$
23: $eligibleMethodList \leftarrow deferredMethodSet$
24: **end if**

25: **return** $eligibleMethodList$

for processing after the complete traversal of the invocation context class hierarchy.

After the complete traversal, if only the set of deferred method binding information is available, we perform a prioritization over the deferred method binding set and return the result.

Algorithm 7 Algorithm for finding method binding list from invocation context

Input: $mName$ is the name of the method, $numberOfArgs$ is the size of the method arguments, $invokerType$ is the type of method invoker, $argTypeList$ is the list of argument types of the method, $invocationContextClassHierarchy$ is an ordered set of invocation context class hierarchy,
Output: $eligibleMethodList$ is the list of eligible method binding list

1: initialize $eligibleMethodList$, $deferredMethodSet$
2: $classNameDeclarationOrderList \leftarrow getClassNameDeclarationOrder()$

3: **while** Iterate over $invocationContextClassHierarchy$ **do**
4: $\quad classSet \leftarrow invocationContextClassHierarchy.get(index)$
5: $\quad eligibleMethodList \leftarrow getQualifiedMethodList(D, J, classSet, mName, numberOfArgs)$

6: $\quad setInvocationContextClassAttribute(eligibleMethodList)$
7: $\quad postModification(eligibleMethodList)$
8: $\quad eligibleMethodList \leftarrow filter(eligibleMethodList, invokerType, argTypeList)$

9: $\quad$ **if** $index! = 0$ **AND** $hasAllDeferredCriteria(eligibleMethodList)$ **then**
10: $\quad\quad deferredMethodSet < -eligibleMethodList$
11: $\quad\quad eligibleMethodList.clear()$
12: $\quad$ **end if**

13: $\quad$ **if** $isNotEmpty(eligibleMethodList)$ **then**
14: $\quad\quad$ **return** $eligibleMethodList$
15: $\quad$ **end if**
16: **end while**

17: **if** $isEmpty(eligibleMethodList)$ **AND** $isNotEmpty(deferredMethodSet)$ **then**
18: $\quad prioritizeDeferredMethods(deferredMethodSet, classNameDeclarationOrderList)$
19: $\quad eligibleMethodList \leftarrow deferredMethodSet$
20: **end if**

21: **return** $eligibleMethodList$

Stage 3: Directly imported class scope In the third stage, we only examine directly imported classes to find the eligible method bindings. Algorithm 8 represents the approach. On line 2, we take a similar approach as in previous stages to find the method bindings based on the method name and number of arguments in the narrowed-down scope of directly imported classes. We also perform post-modification, filtration, and the process of deferring eligible method bindings. However, the process of evaluating deferred method bindings will be conducted after the end of the rest of the

stages. After all the post-processing, if we can find any eligible method binding, we will return it as a result.

Algorithm 8 Algorithm for finding method binding list from directly imported class

---

Input: $mName$ is the name of the method, $numberOfArgs$ is the size of the method arguments, $invokerType$ is the type of method invoker, $argTypeList$ is the list of argument types of the method, $deferredMethodSet$ is the list of deferred methods, $importedClassList$ is the list of classes that is declared directly on import statements

Output: $eligibleMethodList$ is the list of eligible method binding list

1: initialize $eligibleMethodList$
2: $eligibleMethodList \leftarrow getQualifiedMethodList(D, J, importedClassList, mName, numberOfArgs)$

3: $postModification(eligibleMethodList)$
4: $eligibleMethodList \leftarrow filter(eligibleMethodList, invokerType, argTypeList)$

5: **if** $hasAllDeferredCriteria(eligibleMethodList)$ **then**
6: $\quad deferredMethodSet < -eligibleMethodList$
7: $\quad eligibleMethodList.clear()$
8: **end if**

9: **return** $eligibleMethodList$

Stage 4: Inner classes of directly imported class scope In this stage, we will consider all the inner classes of directly imported classes. Algorithm 9 represents this process of this stage. First, we will collect all the inner classes from directly imported classes, represented on line 2. After that, we will find eligible method bindings for those inner classes and perform most modification, filtration and defer criteria check. If we can still find any eligible candidates, we will return.

Stage 5: Imported package scope In this stage, we will move our focus to the imported packages. We will consider all the classes that belong to those packages as candidates for looking at the appropriate method binding. Algorithm 10 presents the process that we perform in this stage. On line 2, we fetch all the eligible method bindings which match the name and number of arguments. On lines 2, 3, and 4, We perform post-modification, filtration, and the process of deferring method bindings.

Stage 6: Super classes of all import classes At this stage, we will explore all the superclasses and interfaces of the directly imported classes to find the eligible method bindings.

Algorithm 9 Algorithm for finding method binding list from inner class

Input: $mName$ is the name of the method, $numberOfArgs$ is the size of the method arguments, $invokerType$ is the type of method invoker, $argTypeList$ is the list of argument types of the method, $deferredMethodSet$ is the list of deferred methods, $importedClassList$ is the list of classes that is declared directly on import statements

Output: $eligibleMethodList$ is the list of eligible method bindings

1: initialize $eligibleMethodList$
2: $innerClassList \leftarrow getInnerClassList(importedClassList)$
3: $eligibleMethodList \leftarrow getQualifiedMethodList(D, J, innerClassList, mName, numberOfArgs)$

4: $postModification(eligibleMethodList)$
5: $eligibleMethodList \leftarrow filter(eligibleMethodList, invokerType, argTypeList)$

6: **if** $hasAllDeferredCriteria(eligibleMethodList)$ **then**
7: $\quad deferredMethodSet < -eligibleMethodList$
8: $\quad eligibleMethodList.clear()$
9: **end if**

10: **return** $eligibleMethodList$

Algorithm 10 Algorithm for finding method binding from imported packages

Input: $mName$ is the name of the method, $numberOfArgs$ is the size of the method arguments, $invokerType$ is the type of method invoker, $argTypeList$ is the list of argument types of the method, $deferredMethodSet$ is the list of deferred methods, $importedPackageList$ is the list of packages declared on import statements

Output: $eligibleMethodList$ is the list of eligible method bindings

1: initialize $eligibleMethodList$
2: $eligibleMethodList \leftarrow getQualifiedMethodList(D, J, importedPackageList, mName, numberOfArgs)$

3: $postModification(eligibleMethodList)$
4: $eligibleMethodList \leftarrow filter(eligibleMethodList, invokerType, argTypeList)$

5: **if** $hasAllDeferredCriteria(eligibleMethodList)$ **then**
6: $\quad deferredMethodSet < -eligibleMethodList$
7: $\quad eligibleMethodList.clear()$
8: **end if**

9: **return** $eligibleMethodList$

We start with immediate parent classes and interfaces. Then, in an iterative process, we traverse their parents until we complete the traversal of the superclass hierarchy. Algorithm 11 presents the approach that we take in this stage. On line 2, we assign directly imported class names to a set. Line 3 shows the iterative process inside which we try to find a set of all immediate parent classes and interfaces and look for eligible method bindings for that set. We again perform the post-modification, filtration, and deferral processes. If we can find any eligible candidates, we will return; otherwise, we will progress through the iterative process. Finally, on line 12, in the case of no eligible method bindings, we check for any deferred method bindings that exist. If there are deferred method bindings, we perform the prioritization and return from the algorithm with deferred method bindings as the result.

Algorithm 11 Algorithm for finding method binding from super classes

Input: $mName$ is the name of the method, $numberOfArgs$ is the size of the method arguments, $invokerType$ is the type of method invoker, $argTypeList$ is the list of argument types of the method, $deferredMethodSet$ is the list of deferred methods

Output: $eligibleMethodList$ is the list of eligible method bindings

1: initialize $eligibleMethodList$
2: $classSet \leftarrow importedClassList$

3: **while** $classSet$ has superclass or interface **do**
4: $classSet \leftarrow getImmediateSuperClassOrInterface(classSet)$
5: $eligibleMethodList \leftarrow getQualifiedMethodList(D, J, classSet, mName, numberOfArgs)$

6: $postModification(eligibleMethodList)$
7: $eligibleMethodList \leftarrow filter(eligibleMethodList, invokerType, argTypeList)$

8: **if** $hasAllDeferredCriteria(eligibleMethodList)$ **then**
9: $deferredMethodSet < -eligibleMethodList$
10: $eligibleMethodList.clear()$
11: **end if**
12: **end while**

13: **if** $isEmpty(eligibleMethodList)$ **AND** $isNotEmpty(deferredMethodSet)$ **then**
14: $prioritizeDeferredMethods(deferredMethodSet, owningClassNameDeclarationOrderList)$
15: $eligibleMethodList \leftarrow deferredMethodSet$
16: **end if**

17: **return** $eligibleMethodList$

Post Modification Process: In each of the 6 stages, we performed post-modification to the eligible method bindings. We provide post-modifications for three different scenarios.

- **Method Invocation of an Array or Varargs:** In the Java programming language, an array is an object and the array type has members such as *length*: a field instance, *clone*: a public method. The return type of the *clone* method should be also the same type as the invoker of the method. However, the byte-code representation provides *java.lang.Object* as the return type for method *clone*. We perform a modification to return the type of array as return type. Java also considers varargs similar to an array. Hence, we also perform similar modifications for varargs.

- **Inclusion of Internal Dependency Property:** In our method binding object model we provide an additional property named *internalDependency*. The value is *boolean*. This property indicates whether this method binding is declared inside the project or declared on external third-party dependencies. The java core packages also fall under external dependency. Based on the list of artifacts that we collect in the earlier stage 4.1, we can resolve this property.

- **Removal of Arguments added in Byte-Code Representation for Inner Class Constructor:** We extract the method binding information from the byte-code representation. The Java compiler, during byte-code generation, performs manipulations on source code, such as adding an outer class as a reference to the non-static inner class constructor if that wasn't explicitly defined. This kind of addition of arguments will create an issue during argument comparisons. To identify and remove the argument that is added by compilers, we depend on the arguments from the invocation context. We take into consideration the first argument of the invocation context. If the first argument is a reference to the outer class, we will consider the first argument of the method as explicitly defined. Otherwise, we will consider the first argument as a post-addition by the java compiler and remove the argument from the argument list of method binding. We will also perform the number of arguments check for all eligible method bindings after the removal of arguments added in byte-code representation.

Filtration Process: In this section, we explain the filtration process that we perform during the resolution of appropriate method binding. We have listed down all the different criteria for filtration as well as explained each criteria-based filtration process.

Filtration based on Invoker Type: This filtration will be applicable for method invocations that have an invoker type. The appropriate method binding may belong to the invoker type class or any superclass or interface. So, in this filtration process, we compare the class name collected from the invoker type with the class name collected from method binding. The method binding contains the class name where the method is declared. We will consider all the bindings eligible if the class name of the invoker type or the name of any parent class or interface matches the class name of the method binding. Algorithm 12 shows the filtration process.

In line 1, we collect the qualified class name from the invoker type. In line 2, we create a map from *eligibleMethodBindingList* where the key is the class name of the method binding and the value is the list of method bindings that have the same class name. On line 3, we also collect all the class names from all the method bindings.

In line 5, we try to compare the class name of the invoker type with all the class names of the method bindings. If we can find a match, we will consider that method binding eligible. Otherwise, we will check whether *invokerType* is an array, and if that is true, we will consider all method bindings that have the class name *java.lang.Object*. Because all array instances have *java.lang.Object* as their parent class. For eligible method bindings, we also assign a matching distance, which will be used later to prioritize in the case of multiple candidates.

In the case of the unavailability of eligible method bindings that match the invoker type, we will move on to the next stage, where we will iterate over the parent classes and interfaces of the invoker type and try to find a match for eligible method bindings. We will also maintain a variable *distance* which will preserve a score that will increase in the case of moving from the most immediate parent classes to further parent classes in the class hierarchy. On line 13, we start the iteration process, and on line 14, we collect all the immediate parent classes and interfaces of the invoker class and also increment the distance on the next line. On line 16, we iterate over all the classes that we have found from method bindings and check if we can find any matching between the parent classes and the class from the method binding. If we can find any matches, we can assign the matching

distance score and add the method binding to the eligible list. On line 22, we perform the process of deferring eligible method bindings based on the deferred criteria. We hold the list of deferred eligible method bindings in a separate list until the traversal is complete. After the deferral process, on line 26, we check the availability of eligible method bindings. If any eligible method binding is available, we can exit the iteration process. Otherwise, we fetch the parent classes and iterate the process until the superclass hierarchy traversal is complete.

After the iteration process, if we have any eligible bindings, we will return the result. Otherwise, we will return the deferred method binding list.

Filtration based on Method Argument Types: Filtration based on method argument types constitutes identifying eligible method binding candidates based on the comparison between arguments collected in the invocation context and arguments from the eligible method bindings. We will narrow down the eligible method binding list based on the successful matching of all the method parameters with their corresponding arguments from the invocation context.

Usually, we consider the comparison a successful match if the type class name of the argument passed in the invocation context matches the type class name of the argument in the method binding. However, the Java programming language offers features such as primitive type narrowing, primitive type widening, conversion of primitive to wrapper classes, and passing variadic arguments. Therefore, these language features need to be considered during the comparison of arguments. For overloaded method declarations, we will find multiple method binding candidates where we have to choose the appropriate one based on invocation context arguments. When the argument type class names do not match directly and we need to perform some form of conversion or inference in order to be considered as matching, we assign a distance value in those scenarios. The distance represents the closeness of matching between the argument of the invocation context and the argument of that particular method binding. Later, we will explain the assignment of the distance in detail.

Algorithm 13 shows the approach that we take in order to identify eligible method binding candidates. The algorithm takes the list of arguments of the method binding, the list of arguments collected from the invocation context, and the method binding as input and produces a boolean value indicating the candidacy of eligibility for that particular method binding.

For any method references and lambda expressions, we create a type representation named

Algorithm 12 Algorithm for filtration process based on invoker type

Input: $eligibleMethodBindingList$ is the list of eligible method bindings, $invokerType$ is the type of method invoker, $isSuperInvoker$ represents whether invoker is *super* keyword
Output: $filteredMethodBindingList$ is the list of eligible method bindings

```
1: invokerClassName ← invokerType.getQualifiedClassName()
2: Map < String, List < MethodBinding >> methodBindingListByClassNameMap ←
   getMethodBindingListByClassNameMap(eligibleMethodBindingList)
3: methodBindingClassNameList ← methodBindingListByClassNameMap.keys()
4: initialize filteredMethodBindingList

5: if methodBindingClassNameList.contains(invokerClassName) AND !isSuperInvoker then
6:     filteredMethodBindingList.addAll(
   methodBindingListByClassNameMap.get(invokerClassName))

7: else if isArray(invokerType) AND methodBindingClassNameList.contains('java.lang.Object')
   then
8:     filteredMethodBindingList.addAll(
   methodBindingListByClassNameMap.get(invokerClassName))
9:     setInvokerMatchingDistance(filteredMethodBindingList, maxSuperClassDistance)
10: else
11:     initialize classSet, deferredMethodBindingSet,distance = 0
12:     classSet.add(invokerClassName)

13:     while classSet is not empty do
14:         classSet ← getSuperClass(classSet, isSuperInvoker)
15:         distance ← distance + 1

16:         for class in methodBindingClassNameList do
17:             if classSet.contain(class) then
18:                 invocationTypeDistance ←
   (class = "java.lang.Object")?superClassMaxDistance : distance
19:                 setInvokerMatchingDistance(filteredMethodBindingList,
   invocationTypeDistance)
20:             end if
21:         end for

22:         if hasDeferredCriteria(filteredMethodBindingList) then
23:             deferredMethodBindingSet.addAll(filteredMethodBindingList)
24:             filteredMethodBindingList.clear()
25:         end if

26:         if filteredMethodBindingList is not empty then
27:             break
28:         end if
29:     end while

30:     if filteredMethodBindingList is empty AND deferredMethodBindingSet not empty then
31:         filteredMethodBindingList.addAll(deferredMethodBindingSet)
32:     end if
33: end if

34: return filteredMethodBindingList
```

FunctionTypeInfo that contains the types of the arguments and the return type collected from the invocation context. In Java, each lambda expression or method reference represents a functional interface. A functional interface is an interface with only one abstract method, and the argument types and return type of the abstract method must match the argument types and return type of the corresponding lambda expression or method reference. Therefore, we start our filtration process by converting *FunctionTypeInfo* into the functional interface class type representation in line 1. We perform the conversion based on the match of argument types and return type of *FunctionTypeInfo* with its corresponding functional interface argument's abstract method. If we can perform the conversion successfully, we will continue with matching the rest of the arguments. Otherwise, we will consider the method binding ineligible due to the mismatch of arguments and return *false*.

Conversion of FunctionTypeInfo Arguments Algorithm 14 shows the process of conversion of *FunctionTypeInfo* arguments passed during method invocation. We start the process by iterating the argument list and processing each *FunctionTypeInfo* for possible conversion in line 2. For the *FunctionTypeInfo* argument, we assume the corresponding method-binding argument is a functional interface. Therefore, we search for the only abstract method binding information declared on that functional interface in line 7. If we are unable to locate the abstract method, we will consider that the particular method-binding argument is not a functional interface and return *false* confirming the ineligibility of the method binding in terms of argument matching.

We start by iterating over the list of arguments provided in the invocation context. On line 2, we check whether the type of the argument is *FunctionTypeInfo* and start the process of argument matching from there. On line 7, we try to fetch the method binding information of the abstract method from the current method binding argument type. If we cannot find any abstract method, we can consider that particular argument of method binding to be not a functional interface and therefore return *false* so that we can ignore that particular method binding from eligibility consideration.

The *FunctionTypeInfo* argument can be constructed from a constructor reference of a inner class where we may find outer class references as arguments that are added later by the compiler in the byte-code representation to resolve the accessibility of the outer class from the inner class. As a result, we may encounter *FunctionTypeInfo* with an additional number of arguments. The addition

Algorithm 13 Algorithm for filtration process based on method arguments

Input: $argTypeList$ is the list of method parameters passed as arguments, $mBindingArgTypeList$ is the list of argument types of method binding, $mBinding$ is the eligible method binding

Output: $boolean$ represents whether particular $mBinding$ satisfies all constraints

1: $isSuccess \leftarrow conversionOfFunctionTypeInfoArguments(argTypeList, mBindingArgTypeList)$
2: **if** $not(isSuccess)$ **then**
3:     **return** $false$
4: **end if**

5: $removeAllCommonArguments(argTypeList, mBindingArgTypeList)$

6: **if** $isEmpty(argTypeList)$ AND $isEmpty(mBindingArgTypeList)$ **then**
7:     **return** $true$
8: **end if**

9: initialize $matchedArgList$
10: **for** Iterate over each $index$ of $argumentTypeList$ **do**
11:     $arg \leftarrow argTypeList.get(index)$
12:     $mBindingArg \leftarrow mBindingArgTypeList.get(index)$
▷ Primitive Type Conversion
13:     **if** $isPrimitive(arg)$ AND $isPrimitive(methodBindingArg)$ **then**

14:         **if** $processPrimitiveTypeConversion(arg, mBindingArg, mBinding, matchedArgList)$ **then**
15:             **continue**
16:         **else**
17:             **return** $false$
18:         **end if**
19:     **end if**
▷ Primitive To Primitive Wrapper Conversion
20:     **if** $processPrimitiveToWrapperMatching(arg, mBindingArg, mBinding, matchedArgList)$ **then**
21:         **continue**
22:     **end if**
▷ Primitive Wrapper To Primitive Conversion
23:     **if** $processWrapperToPrimitiveMatching(arg, mBindingArg, mBinding, matchedArgList)$ **then**
24:         **continue**
25:     **end if**
▷ NULL To Primitive Conversion
26:     **if** $processNullToPrimitiveComparison(arg, mBindingArg, matchedArgList)$ **then**
27:         **continue**
28:     **end if**
▷ Varargs Argument
29:     **if** $processVarargsArgument(argTypeList.subList(index), mBindingArg, mBinding, matchedArgList)$ **then**
30:         **break**
31:     **end if**

▷ Object Array To Object Wrapper Conversion

32: **if** $processArrayOfObject(arg, mBindingArg, matchedArgList)$ **then**
33: **continue**
34: **end if**

▷ Array Dimension

35: **if** $checkMismatchOfArrayDimension(arg, mBindingArg, matchedArgList)$ **then**
36: **return** $false$
37: **end if**

▷ Special Type

38: **if** $processArgMatchingWithSpecialType(arg, mBindingArg, mBinding, matchedArgList)$ **then**
39: **continue**
40: **end if**

▷ Super classes and interfaces

41: $processTraversalOfSuperClasses(arg, mBindingArg, mBinding, matchedArgList)$
42: **end for**
43: $processReducedArgsForVarargs(argList, mBindingArgList, matchedArgList, mBinding)$
44: $removeAllCommonArguments(mBindingArgList, matchedArgList)$
45: **return** $isEmpty(mBindingArgList)$

of outer class references only happens when references were not explicitly declared. So on line 11, we perform the reduction of arguments from *FunctionTypeInfo* for inner class constructors if we find any mismatch in size between the abstract method's arguments and arguments of *FunctionTypeInfo*.

In line 13, we perform the comparison between the arguments of the abstract method and *FunctionTypeInfo*. Upon a successful match, we replace the *FunctionTypeInfo* with its corresponding functional interface type collected from method binding on line 14. Otherwise, we will return *false*.

After the conversion of *FunctionTypeInfo* arguments, we try to directly match the qualified class names of arguments provided from invocation context and method binding. We remove the matched class names from both lists as we consider those argument type as same. On line 5 of algorithm 13, we perform the removal operation based on the class name match of arguments. After the removal operation, if both argument lists are empty we can consider the method binding as eligible and return *true*. Otherwise, we will move on to our next steps.

As part of our next step, we iterate over each argument of invocation context and its corresponding argument of method binding and perform various conversion operations to check whether both arguments successfully match. We also assign a distance value for each argument of the method binding if the matching is considered to be inferred. By default, the distance value will always be

Algorithm 14 Conversion of FunctionTypeInfo

```
1:  function CONVERSIONOFFUNCTIONTYPEINFOARGUMENTS(argTypeList,mBindingArgTypeList)
2:      for i = 0; i < argTypeList.size(); i++ do
3:          arg ← argTypeList.get(i)
4:          if arg.isFunctionTypeInfo() then
5:              functionTypeInfo ← (FunctionTypeInfo)arg
6:              mBindingArg ← mBindingArgTypeList.get(i)
7:              mBindingList ← getAbstractMBindingOfFuncInterface(mBindingArg.getClassName())

8:              if isEmpty(mBindingList) then
9:                  return false
10:             end if

11:             reduceArgumentsForInnerClassConstructor(arg, mBindingList)

12:             matched ← false
13:             if isArgumentsMatched(functionTypeInfo, mBindingList.first() then
14:                 argTypeList.set(i, mBindingList.first())
15:                 matched ← true
16:                 break
17:             end if

18:             if not(matched) then
19:                 return false
20:             end if
21:         end if
22:     end for

23:     return true
24: end function
```

zero. The zero distance value means both arguments are identical. The aggregated distance of all the arguments of the method binding represents how closely the method binding's arguments matched with provided arguments from the invocation context. We initialize a list *matchedArgList* on line 9 to keep track of all the matched arguments.

Primitive Type Conversion: As a part of specific scenarios, we begin by determining if the argument from invocation context and the argument from method binding are primitive types. If the check holds true, we perform the argument matching based on primitive type conversion. If after the conversion we can find a match, we move on to the next argument matching. Otherwise, we consider that method binding ineligible due to argument mismatch and return *false*. In Java, there are two different approaches in which a different primitive type can be passed against another primitive type argument defined in the method declaration.

Widening Primitive Conversion: This conversion allows smaller size primitive types (i.e., int) to be passed to bigger primitive types (i.e., double, long, double). Table 4.2 shows the type and their corresponding allowed conversion types. Also, during this conversion, a precedence order is maintained. The precedence order is based on the size of the type. For example, for `int`, the precedence order will be `long, float, double` where `long` will have priority over `float, double`. Listing 2 shows an example of the precedence order of widening primitive conversion. The code snippet is extracted from JFreechart[3]. We have enlisted both method invocation context and method declaration context in this snippet. We want to focus on method invocation *moveTo* (line 20) with two *int* arguments. But in the method declaration context, one method declaration has two *double* arguments and the other method declaration has two *float* arguments. Java compiler will link the invocation to the method declaration with two *float* arguments (line 34).

Narrowing Primitive Conversion: This conversion allows primitive types with larger values to be passed to smaller primitive method arguments. As a result, this type of conversion may lose information about the magnitude of numeric value, precision, and range. Table 4.3 shows the types and allowed conversion types.

Algorithm 15 shows the approach where we perform the primitive conversion check and based on the matching we also assign the distance value. We have assigned two different distance values

[3]https://github.com/jfree/jfreechart

Table 4.2: Eligible Widening Primitive Type Conversions

Type	Allowed Conversion Types
byte	short, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Table 4.3: Eligible Narrowing Primitive Type Conversions

Type	Allowed Conversion Types
short	byte, char
char	byte, short
int	byte, short, char
long	byte, short, char, int
float	byte, short, char, int, long
double	byte, short, char, int, long, float

for these two conversions. The distance value of widening primitive type conversion is less than the distance value of narrowing primitive type conversion. Due to the possible loss of information, we will always prefer widening type conversion over narrowing type conversion. On lines 5 and line 9, we add the matched argument to the list which helps us to keep track of the matched arguments. The distance value for widening primitive conversion will also consider the precedence order. We add the precedence order as a fraction to the *primitiveWideningDistance* on line 3.

Primitive to Primitive Wrapper Conversion: Generics language feature of Java does not allow to pass primitive types as type arguments. Hence, object wrapper classes were introduced to encapsulate primitive types and these primitive wrapper classes are used in those scenarios. Table 4.4 shows the primitive types and their corresponding wrapper classes. Java has also introduced autoboxing and unboxing features. In the autoboxing feature, java allows automatic conversion from primitive types to their corresponding object wrapper class. On the other hand, the unboxing feature allows the automatic conversion from the primitive wrapper class to the primitive types.

During argument comparison, we also consider the conversion of primitive types to their corresponding object wrapper class. Algorithm 16 shows the approach of checking the conversion. On

Listing 2 Example of Precedence Order of Widening Primitive Conversion

```
1
2  //Method Invocation Context
3  public class PinNeedle extends MeterNeedle implements
4                                               Cloneable, Serializable {
5   ...
6   @Override
7      protected void drawNeedle(Graphics2D g2, Rectangle2D plotArea,
8              Point2D rotate, double angle) {
9          ...
10         int maxY = (int) (plotArea.getMaxY());
11         int midX = (int) (plotArea.getMinX() +
12                                 (plotArea.getWidth() / 2));
13         //int midY = (int) (plotArea.getMinY() +
14                                 (plotArea.getHeight() / 2));
15         int lenX = (int) (plotArea.getWidth() / 10);
16         if (lenX < 2) {
17             lenX = 2;
18         }
19         pointer.moveTo(midX - lenX, maxY - lenX);
20         ...
21     }
22  }
23
24  // Method Declaration Context
25 public abstract class Path2D implements Shape, Cloneable {
26  ...
27     public final synchronized void moveTo(double x, double y) {
28         if (numTypes > 0
29             && pointTypes[numTypes - 1] == SEG_MOVETO) {
30             ....
31     }
32
33     public final synchronized void moveTo(float x, float y) {
34         if (numTypes > 0
35             && pointTypes[numTypes - 1] == SEG_MOVETO) {
36             ...
37     }
38  ...
39  }
```

line 2, we check whether the provided argument is primitive and fetch the wrapper class from the primitive type and try to compare it with the argument of method binding. If we can find a match,

Algorithm 15 Matching Based on Primitive Type Conversion

```
1: function PROCESSPRIMITIVETYPECONVERSION(arg,mBindingArg,mBinding,argList)
2:     if isPrimitiveWidening(arg, mBindingArg) then
3:         distance ← primitiveWideningDistance
   +(0.1 * (getPriorityOrderedPrimitiveList(arg).indexOf(mBindingArg) + 1))

4:         setArgumentMatchingDistance(mBinding, distance)
5:         argList.add(mBindingArg)

6:         return true
7:     else if isPrimitiveNarrowing(arg, mBindingArg) then
8:         setArgumentMatchingDistance(mBinding, primitiveNarrowingDistance)
9:         argList.add(mBindingArg)

10:        return true
11:    else
12:        return false
13:    end if
14: end function
```

Table 4.4: Primitive Types and their corresponding Wrapper Class

Primitive Type	Primitive Wrapper Class
`byte`	`java.lang.Byte`
`short`	`java.lang.Short`
`char`	`java.lang.Character`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`
`boolean`	`java.lang.Boolean`

we will consider it a valid conversion and add the argument to the list of already matched argument list on line 5. On line 3, we also assign matching distance defined as *primitiveToWrapperDistance*.

Algorithm 16 Matching based on Conversion of Primitive to Primitive Wrapper

1: **function** PROCESSPRIMITIVETOWRAPPERCONVERSION(arg,$mBindingArg$,$mBinding$, $argList$)
2: **if** $isPrimitive(arg)$
AND $getPrimitiveWrapperClass(arg).equals(mBindingArg.getClassName())$ **then**
3: $setArgumentMatchingDistance(mBinding, primitiveToWrapperDistance)$
4: $argList.add(mBindingArg)$

5: **return** $true$
6: **end if**

7: **return** $false$
8: **end function**

Primitive Wrapper to Primitive Conversion: To support the primitive unboxing feature we also perform the comparison of arguments based on a conversion from primitive object wrapper class to primitive. Algorithm 17 shows the function where we perform a conversion check. On line 2, we check whether the argument from method binding is a wrapper class and compare the primitive type that we get from the wrapper class using information from table 4.4 and argument collected from invocation context. If we find the match we will consider a valid conversion, assign a match distance *wrapperToPrimitiveDistance* and add it to the list of matched arguments. We return *true* if we find any valid conversion match and move to the next argument. Otherwise, we will proceed to the next matching criteria.

Algorithm 17 Matching based on Primitive Wrapper to Primitive Conversion

1: **function** PROCESSWRAPPERTOPRIMITIVEMATCHING(arg,$mBindingArg$,$mBinding$, $argList$)
2: **if** $isPrimitiveWrapper(mBindingArg)$
AND $getPrimitive(mBindingArg).equals(arg.getClassName())$ **then**
3: $setArgumentMatchingDistance(mBinding, wrapperToPrimitiveDistance)$
4: $matchedArgumentList.add(mBindingArg)$

5: **return** $true$
6: **end if**

7: **return** $false$
8: **end function**

Matching based on Primitive to Special Type Class Conversion: Primitive types are also allowed a few specific type conversions. For a *java.lang.Object* or a *java.lang.Comparable* argument of a declared method, we can pass primitive as value. For numeric primitive types such as int, long, and double, it is also allowed to be passed as value if the argument of the declared method is *java.lang.Number*. Algorithm 18 shows these arguments checking and assignment of distance based on matching.

Algorithm 18 Matching based on Primitive to Special Type Class Conversion

1: **function** PROCESSPRIMITIVETOSPECIALTYPEMATCHING(arg,$mBindingArg$, $mBinding$,$argList$)
2: **if** $isPrimitive(arg)$ **then**
3: **if** "$java.lang.Object$"$.equal(mBindingArg.getClassName())$ **then**
4: $setArgumentMatchingDistance(mBinding, primitiveToObjectDistance)$
5: $matchedArgumentList.add(mBindingArg)$

6: **return** $true$
7: **else if** "$java.lang.Comparable$"$.equals(mBindingArg.getClassName())$ **then**
8: $setArgumentMatchingDistance(mBinding, primitiveTocomparableDistance)$
9: $argList.add(mBindingArg)$

10: **return** $true$
11: **end if**
12: **else if** $isPrimitiveNumericType(arg)$
AND "$java.lang.Number$"$.equal(mBindingArg.getClassName())$ **then**
13: $setArgumentMatchingDistance(mBinding, primitiveNumericToNumberDistance)$
14: $argList.add(mBindingArg)$

15: **return** $true$
16: **end if**

17: **return** $false$
18: **end function**

Null To Primitive Conversion: In Java, *null* is a special type that does not have a name and hence cannot be declared as a variable. The expression of Null type is *null*. Null type can be assigned to any reference type. Java allows a few special behaviors for Null type such as we can pass Null type as a value for any primitive method argument. To accommodate this behavior, we have implemented this function 19. On line 2, we check the argument that is provided from the invocation context is of Null type and the argument of method binding is of primitive type. If the check holds true, we

consider this as a valid conversion and add it to our list of already matched arguments.

Algorithm 19 Matching based on Null and Primitive Type Comparison

```
1: function PROCESSNULLTOPRIMITIVECOMPARISON(arg,mBindingArg,argList)
2:     if isNull(arg) AND not isPrimitive(mBindingArg) then
3:         argList.add(mBindingArg)
4:         return true
5:     end if

6:     return false
7: end function
```

Matching of Varargs Argument: Java introduced *varargs* feature to pass an arbitrary number of arguments as method parameters. Varargs can only be used as the final argument of the method. Java compiler considers varargs argument as an array of elements. We try to approach the matching operation of varargs argument of method binding with the list of arguments collected from the invocation context in a different manner. Algorithm 20 shows the approach.

Each method binding has a property named *varargs* which indicates whether that method declaration has varargs argument. On line 2, we check the varargs property and whether the current argument of method binding is an array. If both of these checks hold true, we perform the final argument matching check. The arguments matching check consists of the following constraints. This function considers all remaining arguments *argSubList* collected from the invocation context as candidates for varargs.

(1) No argument of invocation context can have a mismatch of dimension with varargs element type. By default, the non-array type will be considered as zero dimensions.

(2) If qualified names of the argument types of invocation context directly match with the qualified name of the type of varargs argument, we will consider it a successful match. Otherwise, any superclass or interface has to match the qualified class name of varargs type argument. For primitive types, we convert them to their corresponding wrapper classes 4.4, before super class traversal.

Matching based on Array to Object Array Conversion If the argument of the method binding is an array of *java.lang.Object*, we have to process the matching in a different manner. Because an

Algorithm 20 Matching of Varargs Argument

1: **function** PROCESSVARARGSARGUMENT($argSubList$,$mBindingArg$,$mBinding$,$argList$)
2: **if** $mBinding.isVarargs()$ AND $isArray(mBindingArg)$ AND $isArgumentsMatched(mBindingArg, argSubList)$ **then**
3: $setArgumentMatchingDistance(mBinding, varargsDistance)$
4: $argList.add(mBindingArg)$
5: **return** $true$
6: **end if**
7: **return** $false$
8: **end function**

array of any type can also be considered as *java.lang.Object*. So during dimension matching of both arguments, we have to consider the behavior. Algorithm 21 shows the approach where we check whether the argument *mBindingArg* of method binding is an array of *java.lang.Object* and argument *arg* from invocation context also of type array. Based on the result of these two checks, we perform the dimension-matching check on line 2. As any type or array of types can be considered a valid argument, we consider the dimension of the argument would be 1 less than an array of objects. For the non-array type, the dimension will be 0.

Algorithm 21 Matching based on Array to Object Array Conversion

1: **function** PROCESSARRAYOFOBJECT(arg,$mBindingArg$,$argList$)
2: **if** $isObjectArray(mBindingArg)$ AND $isArray(arg)$ AND $matchesDimension(arg, mBindingArg)$ **then**
3: $matchedArgumentList.add(mBindingArg)$
4: **return** $true$
5: **end if**
6: **return** $false$
7: **end function**

Matching Based on Object Array to Object Conversion: Java allows an object array as value to be passed an Object argument of a method. So we can consider this a valid conversion. But we want to add a distance value so that, if there are alternative method binding with the Object Array argument, we can give priority to the alternative solution. Algorithm 22 shows the process of this conversion.

Algorithm 22 Matching Based on Object Array to Object Conversion

1: **function** PROCESSOBJECTARRAYTOOBJECT(arg,$mBindingArg$, $mBinding$,$argList$)
2: **if** "$java.lang.Object$".$equal(arg.getClassName())$ AND "$java.lang.Object$".$equal(mBindingArg.getClassName())$ AND $isArray(arg)$ **then**
3: $setArgumentMatchingDistance(mBinding, objectArrayToObjectDistance)$
4: $argList.add(mBindingArg)$

5: **return** $true$
6: **end if**

7: **return** $false$
8: **end function**

Matching with Super Classes and Interfaces: Java also allows any sub-class type to be passed as a value for method arguments. So during arguments matching between invocation context and method binding, we check all the superclasses and interfaces of the argument of invocation context and try to match with the corresponding argument's class name from method binding. Algorithm 23 shows the iterative process where we start from the class name of the argument of invocation context. We also initialize a distance value with 0. On line 5, we try to find all the parent class and interfaces from that class and check whether any of these class or interface matches the argument class name from method binding. With each iteration, we increment the distance value by 1. If we find any match, we first assign the distance value to the particular method binding. If the matched class name is *java.lang.Object*, we assign *maxSuperClassDistance* which will assign a large value as distance, since we will try to prioritize any other matched argument. Otherwise, the assigned distance value would be our computed distance in each iteration.

Process Reduced Arguments for Varargs: Varags method arguments also allows to send zero arguments. The compiler will internally create an empty array as an argument. To facilitate the feature, we perform the processing of reduced arguments outside of the arguments iteration. Algorithm 24 shows the process where we compare the size of both lists of arguments. If the size of the argument list from the invocation context is one less than the argument list from method binding and also if the method binding property *varargs* is true, we can consider this as a scenario of reduced arguments. We assign a distance value to the method binding and add the last argument of the method binding to the list of matched arguments.

Algorithm 23 Matching Based on Super Class Conversion

1: **function** PROCESSTRAVERSALOFSUPERCLASSES(arg,$mBindingArg$,$mBinding$,$argList$)
2: initialize $classNameSet$, $distance = 0$
3: $classNameSet.add(arg.getClassName())$
4: **while** $isNotEmpty(classNameSet)$ **do**
5: $classNameSet \leftarrow getSuperClassNameSet(classNameSet)$
6: $distance \leftarrow distance + 1$
7: **if** $classNameSet.contains(mBindingArg.getClassName())$ **then**
8: **if** "$java.lang.Object$"$.equals(mBindingArg.getClassName())$ **then**
9: $setArgumentMatchingDistance(mBinding, maxSuperClassDistance)$
10: **else**
11: $setArgumentMatchingDistance(mBinding, distance)$
12: **end if**
13: $argList.add(mBindingArg)$
14: **break**
15: **end if**
16: **end while**
17: **end function**

Algorithm 24 Process Reduced Arguments for Varargs

function PROCESSREDUCEDARGSFORVARARGS($argList$, $mBindingArgList$, $mBinding$)
if $mBinding.isVarargs()$ AND $argList.size() == mBindingArgList.size() - 1$ **then**
$setArgumentMatchingDistance(mBinding, varargsDistance)$
$matchedArgumentList.add(mBindingArgList.last())$

return $true$
end if

return $false$
end function

After performing all the remaining argument conversion matching, we compare the remaining list of arguments from method binding with the list of matched arguments. We remove all the arguments which are common between these two lists. Finally, if the argument list from method binding still has any element, we will return *false* as we could not match all the arguments otherwise we will return *true*.

Heuristics of assignment of Distance Value For Method Arguments: The goal of the assignment of distance value for each argument conversion matching is to prioritize the right candidate. For overloaded methods, we may find multiple method binding candidates where arguments of those method binding can be matched with arguments from invocation context using any above-mentioned conversion. Listing 3 shows an example of the selection of method binding based on argument matching distance. In the invocation context, *hashCode* method invocation (line 12) is the focus of our attention. The first argument is *int* and the second argument is of type *java.awt.Color*. On the other hand, on the method declaration side, we are showing two of the most eligible method declaration candidates among others. For both method declarations, the first argument is *int*. But we can also infer both second arguments for *java.awt.Color*. *java.awt.Pain* is an interface that is implemented by *java.awt.Color* class and *java.lang.Object* is the superclass of all java classes. Here, the compiler will always prioritize method binding with *java.awt.Pain* over *java.lang.Object*. So, in order to prioritize the eligible method binding and find the appropriate one, we have introduced argument closeness distance heuristics. We assign all the parent classes and interfaces with distance value 1 as they are the closest in the class hierarchy and increment the distance value with the traversal of superclasses and interfaces in the class hierarchy. For *java.lang.Object* we assign a maximum distance (1000) value as any other eligible parent class will always have priority over *java.lang.Object*. The reason for choosing 1000 as the value is that it is very highly likely that we will find a superclass hierarchy with a depth of 1000. For other special types, table 4.5 shows the distance value. For varargs argument matching we also assign a value that is higher than *java.lang.Object* as we want to prioritize method binding with *java.lang.Object* over varargs of *java.lang.Object*.

Filtration based on Type Instantiation: In this section, we can filter only class constructor

Listing 3 Example of Choosing Appropriate Method Binding based on Argument Matching Distance

```
1
2   //Method Invocation Context
3   public class DefaultShadowGenerator implements ShadowGenerator,
4                                                   Serializable {
5       ...
6       private Color shadowColor;
7       ...
8       @Override
9       public int hashCode() {
10          int hash = HashUtilities.hashCode(17, this.shadowSize);
11          ...
12          hash = HashUtilities.hashCode(hash, this.shadowColor);
13      }
14  }
15
16  // Method Declaration Context
17  public class HashUtilities {
18  ...
19
20      public static int hashCode(int pre, Paint p) {
21          return 37 * pre + hashCodeForPaint(p);
22      }
23
24      public static int hashCode(int pre, Object obj) {
25          int h = (obj != null ? obj.hashCode() : 0);
26          return 37 * pre + h;
27      }
28  ...
29  }
```

methods if we are trying to resolve *ClassInstanceCreation* AST node. We store an additional property named *isClassInstantiation* when we are processing *ClassInstanceCreation* AST node. We check this property before filtering out class constructor methods.

Filtration of Private Methods: In the previous stages, we added an additional attribute named *invocationContextClasAttribture*. This attribute represents whether the particular method binding belongs to the invocation context class. If the value of the attribute is *false*, we can remove all the bindings with the private modifier from considerations.

Filtration based on Abstract Modifiers: In this stage of filtration, if we have more than one

Table 4.5: Conversion Type and their corresponding Distance

Type	Distance
Widening Primitive Type Conversion	1
Narrowing Primitive Type Conversion	2
Primitive to Primitive Wrapper Conversion	1
Primitive Wrapper to Primitive Conversion	1.5
Primitive to Comparable Conversion	1
Primitive Numeric to Number Conversion	1
Primitive to Object Conversion	1
Object Array to Object Conversion	1
Max Super Class	1000
Varargs	1001

eligible method binding and we have at least one method binding with the non-abstract modifier, we will collect all the eligible method bindings which do not have the abstract modifier.

Prioritization based on Argument Matching Distance: In this stage, we will process filtration based on argument matching distance. In the previous method argument matching stage 13, we assign a distance value based on the matching of provided method argument and method binding arguments. We have already explained the heuristics of the assignment of the closeness distance value. In this phase, if we have more than one eligible candidate, we will consider method binding with the minimum argument matching score as the eligible candidate.

Deferring Process of Method Binding Information: During the resolution of appropriate method binding, we may find method binding information which may partially satisfy all the constraints, however, there is a possibility to find a more accurate match. We consider such method bindings as deferred method bindings. During traversal, we store the deferred method bindings in a separate list and continue the traversal process. After complete traversal, if we don't have any better fit, we will consider deferred method binding as the appropriate method binding. We have identified one of three such criteria which makes a method binding eligible for the deferred process.

(1) If the method is an abstract method we can consider it as deferred method binding. Because non-abstract method declaration will have priority over abstract method.

(2) If the method binding is declared in class *java.lang.Object*, we also consider that method

binding eligible for deferring.

(3) The method binding has *argumentMatchingDistance ¿ 0* that represents we have performed any form of inference to perform the argument matching. In this scenario, we will also consider the method binding as a deferred method binding.

In algorithm 6 we perform the deferring criteria check on line 11 and move the deferred method binding to a separate list. After complete traversal, if we still don't have any eligible method binding we try to prioritize deferred method bindings on line 22 and return the deferred method binding list as eligible method bindings.

Prioritization of deferred Method Bindings: During the prioritization process of deferred method bindings, we first narrow down the method bindings with a minimum argument matching distance. As the second step, we perform the filtration process where we only take method bindings with minimum invocation matching distance.

4.2.3 Post-Processing of Method Binding Information:

After identifying the appropriate method binding candidate, we perform post-transformation that will resolve the missing properties collected from byte-code de-compiled representation.

Resolution of Formal Type Parameter:

The generics were introduced on Java 5 in order to ensure type safety during compile time. Generics force the user to define the type during the object instantiation. We can create generic classes or generic methods. Therefore, during the resolution of method binding information of a method invocation, we consider the generics functionality of the method and populate all the arguments and return type with the appropriate type parameter. In this section, we will explain the process of resolving type parameters for the eligible method binding.

In order to get method binding information, we extract all classes, fields, and, methods from the archive jar using a byte-code decompiler. In order to keep backward compatibility with the previous version of java prior 5, the byte-code decompiler stores the generics metadata in a separate field named *signature*. Table 4.6 shows an example of generic types and their corresponding signatures.

Table 4.6: Type With Generic Signature

Type	Signature
List<E>	Ljava/util/List<TE;>;
List<? extends Number>	Ljava/util/List<+Ljava/lang/Number;>;

Each class declaration with a formal type parameter or a method declaration with a formal type parameter will have its corresponding generic type representation. For example, the class signature of a class declared as:

```
C<E> extends List<E>
```

will be:

```
<E:Ljava/lang/Object;>Ljava/util/List<TE;>;
```

In this class signature, <E:Ljava/lang/Object;> represents the formal type parameter E where base type is java.lang.Object. The signature does not contain the name of the class. The signature only contains the parent class or interface with a type parameter.

Also, for the following method with formal type parameter T:

```
<T> Class<? extends T> m (int n)
```

the method signature would be:

```
<T:Ljava/lang/Object;>(I)Ljava/lang/Class<+TT;>;
```

The return type of the method is a generic class java.lang.Class with formal type parameter T. In the signature <T:Ljava/lang/Object;> represents the formal type with base type java.lang.Object. (I) represnts the only method argument int.

The formal type parameters are replaced with type arguments during the instantiation of the class or method invocation. In this section, we will explain the process of extracting the type arguments from invocation and replacing the formal type parameters in method binding.

The resolution of formal type parameters can depend on 1) the type arguments of the method invoker, 2) the type arguments passed during the method invocation, 3) the type of the arguments,

and 4) the type of the return type. We take a precedence-based approach to resolve all the formal type parameters of method binding.

(1) First, we try to check whether method invoker is a parameterized type and type arguments are provided. We store all the type arguments against their corresponding formal type parameter name.

(2) In the next step, if type arguments are provided during method invocation, we collect the list of formal type parameters from the method binding and store the type arguments against their corresponding formal type parameter name. Otherwise, we try to resolve the type argument from method arguments. For example, if the argument from method binding is an instance of a formal type parameter and the argument that we collect from the invocation context is a non-formal type parameter, we can assume the argument from the invocation context is a replacement given that the formal type parameter is not defined in the declared class. We store the invocation context argument against the formal type parameter name defined as the argument from method binding.

(3) We compare the return types between invocation context and method binding, similar to argument comparison, in order to resolve the type replacement of the formal type parameter. We will store the replacement type against the formal type parameter name collected from method binding.

(4) If the method invoker is the invocation context class itself, we traverse through all the method binding arguments and return type and check whether the argument itself is a formal type parameter. If any are found, we will store the argument as a replacement type against the argument's type parameter name.

After extraction of all type parameter names and their corresponding replacement types, we traverse the generic signature of the method binding and replace all the type parameters with the replacement types.

4.2.4 Conversion of variadic Argument:

We consider variadic argument (also known as varargs) as an array of elements with one dimension during our identification. If the particular method binding candidate has *varargs* property. We convert the last argument of that method binding to variadic type representation. Conversion to variadic type will give users appropriate and accurate information about the type of the arguments of the method binding.

Chapter 5

Implementation

In this chapter, we will provide an explanation of the API based on our approach. We will also explain the storage system. We will also introduce the chrome extension that we have developed for GitHub where we display the method signature from the method invocation expression. We will also compare our result with GitHub's own code navigation result.

5.1 "API Finder" API

We have implemented our approach as an Application Programming Interface (API) [17]. According to our methodology, We have also designed our API in two stages. In the first stage, we identify the project java version as well as all the dependent artifacts. We load all the class files in our system based on java and dependent artifact archives. Table 5.1 shows the method summaries of *TypeInferenceAPI* class for loading the Java version and external dependencies. We support both local and remote GitHub public project repositories for artifact retrieval.

For the second stage, Table 5.2 shows the method summaries from *TypeInferenceV2API* class for resolving method binding information. We take all method invocation AST nodes 3.1 as input for identifying their corresponding method binding information.

Listing 4 shows the API usage example of resolving method binding information. The invocation of *loadJavaAndExternalJars* should be performed once per commit of the project. The line 14 shows *loadJavaExternalJars* method takes the commit-id, project name, and project's Github clone

Table 5.1: API for Loading Project Java and Dependent Artifacts [*TypeInferenceAPI* Class]

Modifier & Type	Method & Description
static Tuple2<String, Set<Artifact>>	loadJavaAndExternalJars(String commitId, String projectName, String cloneUrl) Returns the java version and external dependency artifacts of the project hosted on remote repository
static Tuple2<String, Set<Artifact>>	loadJavaAndExternalJars(String commitId, String projectName, Git git) Returns the java version and external dependency artifacts of the project hosted on local repository

Table 5.2: “API Finder” API [*TypeInferenceV2API* Class]

Modifier & Type	Method & Description
static MethodInfo	getMethodInfo(Set<Artifact> dependentArtifactSet, String javaVersion, MethodInvocation methodInvocation) Returns method binding from method invocation
static MethodInfo	getMethodInfo(Set<Artifact> dependentArtifactSet, String javaVersion, SuperMethodInvocation superMethodInvocation) Returns method binding from super method invocation
static MethodInfo	getMethodInfo(Set<Artifact> dependentArtifactSet, String javaVersion, ClassInstanceCreation classInstanceCreation) Returns method binding from class instance creation
static MethodInfo	getMethodInfo(Set<Artifact> dependentArtifactSet, String javaVersion, ConstructorInvocation constructorInvocation) Returns method binding from constructor invocation
static MethodInfo	getMethodInfo(Set<Artifact> dependentArtifactSet, String javaVersion, SuperConstructorInvocation superConstructorInvocation) Returns method binding from super constructor invocation

Listing 4 Example of API Finder Usage

```java
public class APIFinderUsage {
    public static void getMethodBindingFromMethodInvocation() {
      /*
       *   Stage 1: loading project's Java and External
       *   Dependency Artifacts
       */
      String projectName = "kubernetes-client";
      projectUrl =
      "https://github.com/fabric8io/kubernetes-client.git";
      commitId = "43af167c663031dac37f08dda36e35e512462071";

      Tuple2<String, Set<Artifact>> dependency =
        TypeInferenceAPI.loadJavaAndExternalJars(commitId,
        projectName, projectUrl);
      String filePath =
      "crd-generator/apt/src/main/java/io/fabric8/crd"
      + "/generator/apt/CustomResourceAnnotationProcessor.java";
      // collect source content from remote project repository
      String sourceContent = GitUtil.getFileContentFromRemote(
      filePath, projectUrl, commitId);

      // parse the source code to create CompilationUnit
      CompilationUnit compilationUnit =
      ASTUtils.getCompilationUnit(sourceContent);
      compilationUnit.accept(new ASTVisitor() {
        @Override
        public boolean visit(
          MethodInvocation methodInvocation) {
            /*
             * Stage 2: Identify method binding from any
             * method invocation
             */
             MethodInfo methodInfo = TypeInferenceV2API
            .getMethodInfo(dependency._2(),
            dependency._1(), methodInvocation);

            return true;
        }
      });
    }
}
```

URL as arguments to identify the java version of the project and fetch all the external dependency artifacts and load them into the system. For this example, we want to find method binding information for all the method invocations of class *CustomResourceAnnotationParser*. Therefore, we fetch the source content from GitHub Project Repository using GitHub API on line 20. On line 24 we create *CompilationUnit* from source code content using AST parser. Using the AST parser library, we can traverse all the method invocation inside that *CompilationUnit*. For each method invocation, we are invoking our API to get method binding information on line 35. As arguments, the set of dependent artifacts, the project's Java version, and the method invocation instance is propagated. The API produces the *MethodInfo* object which contains method binding information.

5.2 "API Finder" API Using Fluent Builder Pattern

We have also exposed another version of API where practitioners can send invocation-context information as arguments instead of propagating method invocation AST node. As this API does not require propagating AST Node, practitioners can use any AST parser to parse the source code and send the specific invocation-context information to generate the method binding information. This API is implemented using Fluent Builder Pattern to improve readability and simplify API usage. Table 5.3 shows the method summaries which are available under *TypeInferenceFluentAPI* class. This API requires four required arguments. The arguments can be concatenated to help the identification of appropriate method declaration and resolve the method binding information.

Listing 5 shows the API usage where the only change is in the identification of method binding information. On line 34 we construct the *Criteria* instance based on our invocation context information and perform the method binding resolution. API will send us the appropriate method binding as a list where the first element will be the most probable match based on invocation context information.

Due to the fact that inference of invocation context information is not performed in this API, the latency of generating method binding information is less. However, there are a few limitations of this API.

(1) For Generic methods, type parameters will not be resolved for arguments and return type.

Table 5.3: “API Finder” API Using Fluent Builder Pattern [*TypeInferenceFluentAPI* Class]

Modifier & Type	Method & Description
	Criteria(Set<Artifact> dependentArtifactSet, String javaVersion, List<String> importList, String methodName) Constructs a Criteria Class Instance with all the required arguments
Criteria	setInvokerTypeInfo(TypeInfo invokerTypeInfo) Set method invoker type. Type should be constructed as *TypeInfo*
Criteria	setInvokerClassName(String invokerClassName) Set invoker class name. *TypeInfo* will be constructed from class name
Criteria	setSuperInvoker(boolean isSuperInvoker) Set argument *true* for super invoker type expression
Criteria	setClassInstantiation(boolean isClassInstanceCreation) Set argument *true* if the invocation is a class instance creation
Criteria	setArgumentTypeInfo(int argumentIndex, TypeInfo argumentTypeInfo) Set argument order index and argument type. The argument should be constructed as *TypeInfo*
Criteria	setArgumentType(int argumentIndex, String argumentType) Set argument order index and argument type. The argument class name can be passed
List<MethodInfo>	getMethodList() This method invocation will use the criteria as the argument to generate the appropriate method binding list

Listing 5 Example of “API Finder” Fluent API Usage

```java
public class MethodDeclarationFinderFluentAPIUsage {
    public static void getMethodBindingUsingFluentAPI() {
      /*
       *   Stage 1: loading project's Java and External
       *   Dependency Artifacts
       */
       String projectName = "jfreechart-fx";
       String projectUrl =
       "https://github.com/jfree/jfreechart-fx.git";
       String commitId =
       "35d53459e854a2bb39d6f012ce9b78ec8ab7f0f9";

      Tuple2<String, Set<Artifact>> dependency =
        TypeInferenceAPI.loadJavaAndExternalJars(commitId,
        projectName, projectUrl);

      List<String> imports =
        Arrays.asList("import java.lang.*",
        "import org.jfree.data.xy.*");
      String methodName = "getStartX";
      int numberOfArgs = 2;

      /*
       *   Stage 2: Identify method binding information
       *   using Fluent API
       */
      List<MethodInfo> methodInfoList =
      TypeInferenceFluentAPI.getInstance()
      .new Criteria(dependencyTuple._2(), dependencyTuple._1(),
        imports, methodName, numberOfArgs)
                .setInvokerClassName("AbstractIntervalXYDataset")
                .setArgumentType(0, "int")
                .setArgumentType(1, "int")
                .getMethodList();
    }
}
```

(2) Currently, due to the lack of information regarding inner classes and invocation context class hierarchy, a method declared in the invocation context class hierarchy may not be resolved.

5.3 Storage

We have designed our storage as a relational database. Figure 5.1 shows the entity relationship diagram. We have created 8 core entities to represent all the class data.

(1) Jar: This entity stores the artifact's maven coordinates such as group-id, artifact-id, and version. For java core packages, we store *java* as artifact-id and version of java as version field.

(2) Class: This entity store all the class metadata (e.g., name, package name, access modifiers).

(3) Super-Class-Relation: This entity keeps the relationship between the class and its parent classes and interfaces. We also store *type* which represents whether class or interface and *precedence* keeps the order of declaration of all parent classes and interface during class declaration.

(4) Inner-Class-Name: This entity stores the relationship between class and all inner classes. We store the qualified names of all inner classes.

(5) Field: Field entity stores all field metadata (i.e., name, access modifiers).

(6) Method: Method entity stores all the method data except arguments, and exception class names that are thrown by the method.

(7) Argument-Type-Descriptor: This entity stores the type representation of all arguments of all methods.

(8) Thrown-Class-Name: This entity keeps track of qualified class names of all the exceptions the method throws during its declaration.

In addition, we also introduced an in-memory cache named caffeine [1] for efficient retrieval of information. We have used caching to reduce database query executions. We also cache the method binding information against the method invocation node.

[1] https://github.com/ben-manes/caffeine

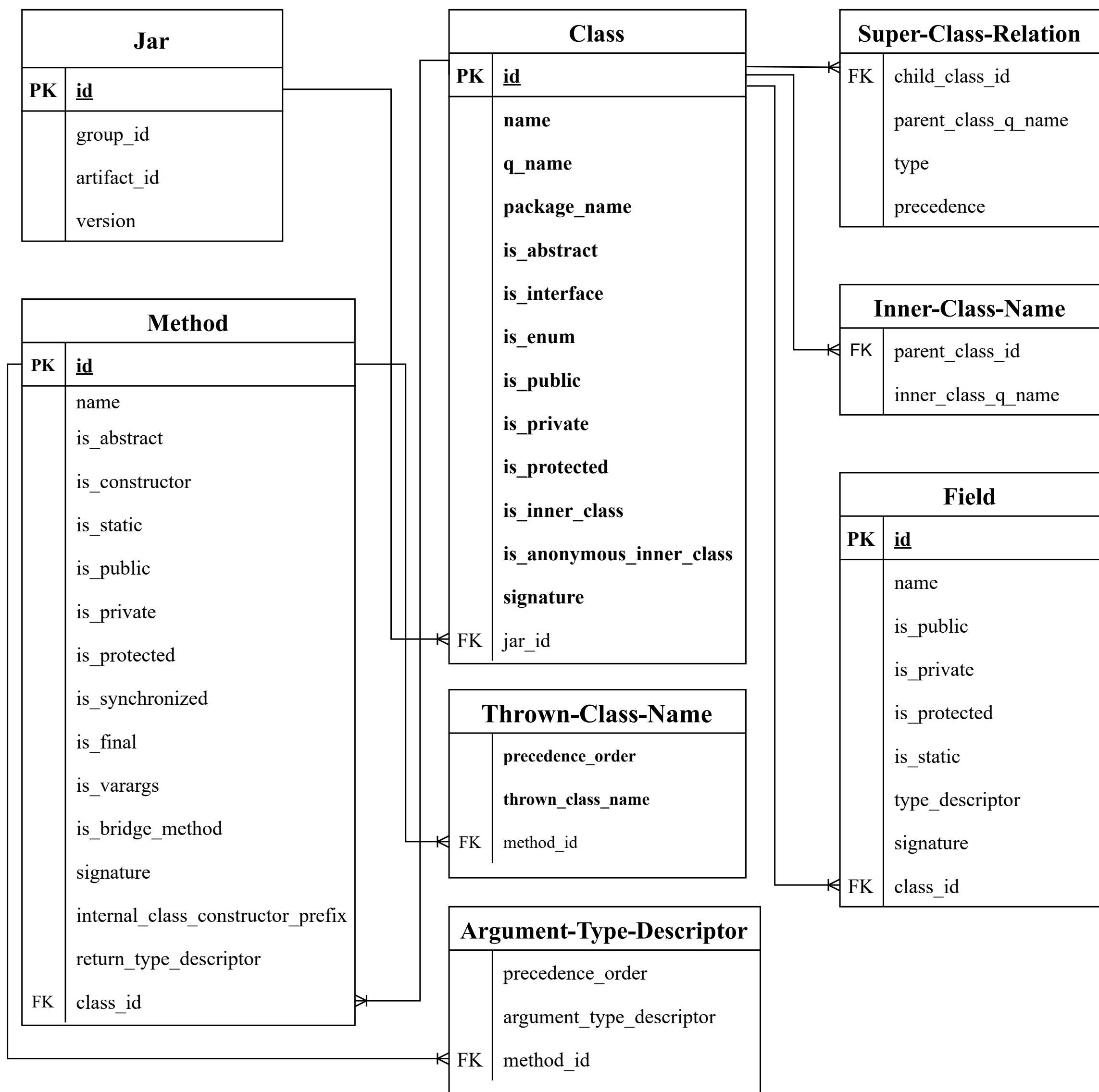


Figure 5.1: Entity Relationship Diagram

```
139        /**
140         * Returns a new list containing the domain crosshairs for this overlay.
141         *
142         * @return A list of crosshairs.
1   java.util.ArrayList :: public void ArrayList (java.util.Collection)
1
145            return new ArrayList<>(this.xCrosshairs);
146        }
147
```

Figure 5.2: “API Finder” Chrome Extension (Collected from JFreeChart [2])

5.4 Chrome Extension for GitHub

As an application of our API, we have developed a google chrome extension named “API Finder” which provides accurate identification of declared method signatures from method references on GitHub. Figure 5.2 shows the suggested method signature from a method reference on GitHub.When a user clicks on any method invocation expression on a GitHub page, we identify the accurate method signature and show the signature in a tooltip. Our Chrome extension consists of two major components.

- **Chrome Extension:** The chrome extension detects the activities of a user (i.e., click) on a Java source file hosted on GitHub and sends information such as page URL, method invocation expression the user clicked on, and line number to the web server. The Chrome extension is also responsible for showing the identified method signature sent by the web server in a tooltip.

- **Web Server:** The web server collects inputs from the chrome extension and constructs the appropriate AST Node and utilizes our API to identify the appropriate method binding information. Finally, construct the method signature from the method binding information and send it to the chrome extension.

[2]https://github.com/jfree/jfreechart/blob/v1.5.3/src/main/java/org/jfree/chart/panel/CrosshairOverlay.java#L145

5.4.1 Comparison between "API Finder" extension and GitHub Code Navigation

GitHub also provides code navigation assistance, When a user clicks on any method invocation expression, GitHub displays a tooltip with a list of possible method declaration instances. However, the identification of the method declaration performed by GitHub code navigation has a few shortcomings.

- **Inability to resolve method references for Java Core Packages and External Libraries**: GitHub code navigation does not provide support for displaying the declared method signature for methods that are declared on Java core packages or declared on classes outside of the project. Figure 5.2 shows such a scenario where for `ArrayList` class instance creation expression, GitHub cannot display the appropriate declared method signature. However, our chrome extension is able to show the declared method signature in such scenarios.

- **No consideration of Invocation Context Information:** GitHub code navigation does not consider invocation context information such as the type of the invoker expression to determine the method declaration as a result, depending on only method name matching will generate inaccurate method declarations. Figure 5.3 displays such an instance where GitHub code navigation provides 16 method definitions and all of them are wrong as they do not consider the type of invocation expression *this.xCrosshairs* which is an instance of *java.util.List*. Our chrome extension considers invocation context data to determine the appropriate method declaration.

- **No consideration of Number of Method Argument:** GitHub code navigation also does not consider the number of arguments for determining method declarations. Figure 5.4 shows such a scenario where there are two overloaded method declaration candidates for *setCircular* method. GitHub code navigation shows both instances of declaration. On the other hand, our chrome extension considers the number of arguments and shows the appropriate declared method signature.

[3]https://github.com/jfree/jfreechart/blob/v1.5.3/src/main/java/org/jfree/chart/panel/CrosshairOverlay.java#L117
[4]https://github.com/jfree/jfreechart/blob/v1.5.3/src/main/java/org/jfree/chart/plot/PiePlot.java#L642

```
115          java.util.List :: public abstract boolean remove (java.lang.Object)
116
117          if (this.xCrosshairs.remove(crosshair)) {
118              crosshair.remove
119              fireOverlayChang
120          }
121      }
122
123      /**
124       * Clears all the domair
125       * {@link OverlayChangeE
126       * were no crosshairs tc
127       */
128      public void clearDomainC
129          if (this.xCrosshairs
130              return;  // noth
131          }
132          for (Crosshair c : g
133              this.xCrosshairs.remove(c);
134              c.removePropertyChangeListener(this);
135          }
136          fireOverlayChanged();
137      }

Definitions    References
Present in 16 files
src/main/java/org/jfree/chart/plot/CombinedDomainCategoryPlot.java
 173    public void remove(CategoryPlot subplot) {
src/main/java/org/jfree/chart/plot/CombinedDomainXYPlot.java
 282    public void remove(XYPlot subplot) {
src/main/java/org/jfree/chart/plot/CombinedRangeCategoryPlot.java
 169    public void remove(CategoryPlot subplot) {
src/main/java/org/jfree/chart/plot/CombinedRangeXYPlot.java
 214    public void remove(XYPlot subplot) {
```

Figure 5.3: No Consideration of Invocation Context (Collected from JFreeChart [3])

```
org.jfree.chart.plot.PiePlot :: public void setCircular (boolean, boolean)
113            setCircular(false, false);
114        }
115
116        /**
117         *
118         *
119         *
120         *

Definitions    References
Present in 1 file
 641    public void setCircular(boolean flag) {
 654    public void setCircular(boolean circular, boolean notify) {
```

Figure 5.4: No Consideration of Number of Method Arguments (Collected from JFreeChart[4])

- **No recognition of constructor reference:** GitHub code navigation also does not display method declaration for constructor reference or super constructor reference. In contrast, our Chrome extension can display the declared method signature for the constructor or the super method reference.

Chapter 6

Evaluation

In this section, we will explain the evaluation process of our approach. In our evaluation, we investigate the following research questions:

- **RQ 1:** What is the accuracy of our proposed approach?
- **RQ 2:** What is the execution time for resolving method binding information?

6.1 Evaluation Setup

6.1.1 Project Selection

The evaluation of our proposed approach is challenging. Only the compiler can generate complete method-binding information from the invocation context. However, the compiler implementation will require the complete project to be compiled and built. Therefore, we have to perform our evaluation on complete projects in order to compare the outcome of our approach against the method binding result generated by the compiler under our evaluation environment. Therefore, we have selected evaluation projects based on a few criteria.

- Build Process: The build process for the project has to be simple and straightforward. We have used Eclipse JDT as our compiler implementation. So, we have to select projects that Eclipse can build properly without any external intervention.

- Project Artifact: Our approach depends on the project's artifact in order to collect all the meta-data from class files. As we want to resolve all the method invocations of the project, we need to provide the project artifact as a dependency so that we can resolve all the internal method invocations. Therefore, for the evaluation, we have to select projects with artifacts published in the public artifact repositories.

- Determination of Release Version: For Gradle projects, contributors can propagate the release version as a command line argument. As a result, the release version defined in the build script may not match the release version of the project in public artifact repositories. It is not possible to collect the artifact of the project from public artifact repositories without an appropriate version. Therefore, we have selected projects where the release version number is specified in the build script.

- Popularity: In the selection of evaluation projects, we also took into account the number of contributors, number of stars, number of releases, activity in projects, and age of the project so that we could select projects that are currently active and well-known.

- Diversity: Diversity also played a key role in the choice of evaluation projects, as we will find diverse language constructs based on the category of the project. For example, a core Java library such as Guava makes extensive use of generics. On the other hand, a chart library such as JFreeChart has more orthodox language usage. Also, projects with different language constructs (e.g., lambda expression, stream API) introduced with different Java versions are ideal candidates for our evaluation.

Based on the aforementioned criteria, 11 open-source projects from various categories have been chosen. The projects that we have chosen for evaluation are listed in Table 6.1. *Kubernetes Client/Java* has the most lines of code among the selected projects, while *ScribeJava* has the fewest. Two projects (i.e., *Mockito* and *JGroups*) have Java version 11, and the rest of the projects have Java version 8. Among all the projects, *JavaParser* has the highest percentage of method invocations with generics and lambda expressions. In terms of build systems, nine projects are Maven-based, and two projects are Gradle based. Our evaluation was conducted on a single version of the project.

Table 6.1: Java Projects for Evaluation

Project Name	Version	Category	Build System	Java Version	LOC	Nested Method Call Percentage	Generics Percentage	Lambda or Method Reference Percentage
JFreeChart	1.5.3	A chart library	Maven	8	264,445	12.66%	1.32%	0%
Guava	30.1.1	A core Java library	Maven	8	339,923	22.27%	26.17%	1.11%
Mockito	4.6.1	A mocking framework for unit test	Gradle	11	78,126	27.26%	12.19%	2.13%
Spring Data JPA	2.7.3	An extension of Spring Data	Maven	8	40,988	26.60%	11.82%	6.45%
JavaParser	3.24.2	An AST Parsing library for Java	Maven	8	154,535	34.79%	25.23%	9.10%
RxJava	3.1.5	Reactive extension for Java	Gradle	8	393,676	12.18%	23.87%	0.03%
Jackson Core	2.14.0-rc2	A data binding library	Maven	8	101,137	8.39%	1.21%	0.03%
Kubernetes Client/Java	15.0.1	Java Client for Kubernetes	Maven	8	1,787,671	25.07%	16.72%	0.34%
Apache Commons IO	2.11.0	A utility library	Maven	8	68,859	15.35%	7.78%	1.31%
ScribeJava	8.3.3	An OAuth Client for Java	Maven	8	17,428	20.53%	2.15%	0.24%
JGroups	5.2.12	A clustering library	Maven	11	135,033	18.12%	2.18%	3.87%

We have attempted to select the most recent versions of the project, as these will contain the most recent version of Java language constructs that are available in the project.

6.1.2 Evaluation Process

We have used the Eclipse JDT Compiler [18] as the compiler implementation for our evaluation. We have designed our evaluation as an Eclipse plugin where we traverse the project using an AST parser and find each method invocation, fetch the method binding generated by the compiler, and compare it with the result of our approach.

To compare the method binding generated from the compiler and the output of our approach, we constructed a generic string representation that contains all the necessary components to identify

the appropriate method declaration. The representation is constructed as:

Qualified-Class-Name::Access-Modifiers Qualified-Class-Name-Return-Type
Method-Name(Qualified-Class-Name-Arg1, Qualified-Class-Name-Arg2)

During the evaluation, we excluded test packages as projects don't include test packages in the project archive. Also, for any anonymous inner class constructor, the compiler considers the anonymous inner class as part of the class where the invocation occurred and generates a method name relative to the position of the inner class. Here is an example of an anonymous inner class constructor 6. For `new SimpleFileVisitor<Path>()` constructor on line 7, the method binding object produced by the compiler will generate the method name as `SourceRoot$1` and don't provide any meaningful information regarding the constructor information from our perspective. So, we also excluded the anonymous inner-class constructors from the evaluation.

Listing 6 Example of Anonymous Inner Class Constructor

```
1
2  public class SourceRoot {
3      ...
4      public List<ParseResult<CompilationUnit>> tryToParse(
5                      String startPackage) throws IOException {
6          ...
7          Files.walkFileTree(path, new SimpleFileVisitor<Path>() {
8              @Override
9          ...
10     }
11 ...
12 }
```

Table 6.2 the distributions of method invocation types for all the method invocations that we have traversed across all the evaluation projects. In all projects, the percentage of *MethodInvocation* is the highest, and *SuperConstructorInvocation* and *ConstructorInvocation* are among the lowest.

6.2 RQ 1: What is the accuracy of our proposed approach?

Motivation This research question assesses the accuracy of our approach. We wish to rely on our strategy to successfully resolve method-binding information for partial programs in all possible

Table 6.2: Distribution of Method Invocation Type

Project Name	Method-Invocation	Class-Instance-Creation	Constructor-Invocation	SuperMethod-Invocation	Super-Constructor-Invocation
JFreeChart	85.3%	11.5%	1.0%	1.5%	0.8%
Guava	87.5%	10.0%	0.2%	1.2%	1.2%
Mockito	81.5%	16.5%	0.5%	0.4%	1.2%
Spring Data JPA	87.9%	9.6%	0.5%	0.9%	1.1%
JavaParser	89.2%	8.6%	0.9%	0.5%	0.9%
RxJava	85.7%	12.5%	0.1%	0.2%	1.5%
Jackson Core	93.4%	4.8%	0.5%	0.4%	0.9%
Kubernetes Client/Java	83.8%	12.4%	2.1%	1.5%	0.2%
Apache Commons IO	79.4%	13.8%	3.6%	1.7%	1.5%
ScribeJava	78.2%	16.4%	2.4%	1.2%	1.7%
JGroups	86.5%	11.6%	0.3%	1.2%	0.4%

scenarios. In our evaluation, successful replication of method binding information generated by the compiler for a complete project will ensure that we will be able to resolve method binding information for method invocation from a partial program, given that we have successfully resolved all the dependent artifacts and Java versions.

Result: Table 6.3 demonstrates the accuracy of our methodology for all evaluation projects. To determine the accuracy, we first determine the number of method invocation instances in which we successfully replicated the compiler-generated method binding information and the number of method invocation instances in which we failed to generate method binding information accurately. The accuracy is the ratio of successful replication instances to the total number of replication instances, including both successes and failures.

$$accuracy = \frac{NumberOfInstancesWithSuccessfulReplication}{TotalNumberOfInstances}$$

JFreechart has the highest success rate with 99.78% accuracy. The JavaParser project, on the other hand, has the lowest success rate with an accuracy of 93.11%. Nine of the eleven projects

Table 6.3: Evaluation of Accuracy

Project Name	Success	Failure		Accuracy
		No Match	Mismatch	
JFreeChart	32,521	50	21	99.78%
Guava	26,579	368	423	97.11%
Mockito	5,892	42	38	98.66%
Spring Data JPA	3,718	141	32	95.55%
JavaParser	35,920	689	1,888	93.31%
RxJava	24,349	26	380	98.36%
Jackson Core	6,917	65	36	98.56%
Kubernetes Client/Java	168,459	675	2,579	98.10%
Apache Commons IO	4,118	57	13	98.33%
ScribeJava	2,072	12	10	98.95%
JGroups	29,202	562	292	97.16%
All Projects	339,747	2,687	5,712	97.59%

have an accuracy rate greater than 97%. In situations where we were unable to accurately match the method binding information, we have divided the occurrences into two categories. *No Match* indicates the number of occurrences in which our approach was unable to extract method-binding information from the method invocation node. In contrast, *Mismatch* represents the number of instances in which generated method-binding information does not match the compiler-generated method-binding information.

For the *Mismatch* failure, we have conducted additional investigations. We divided the number of occurrences of *Mismatch* into five categories. Table 6.4 displays the five different categories and the number of instances that we have identified under each category across all evaluation projects.

- **Class Mismatch:** In this category, we count the number of instances where we have failed to correctly identify the class name of the method-binding information.
- **Return Type Mismatch:** This category indicates the number of occurrences in which we failed to correctly identify the method's return type.
- **Argument Type Mismatch:** This category indicates the number of times we failed to correctly identify the type of each argument.
- **Mismatched Method Modifiers:** This category reflects the number of times we generated

Table 6.4: Mismatch Distribution

Project Name	Class Mismatch	Return Type Mismatch	Argument Type Mismatch	Modifier Mismatch	Throws Exception Mismatch
JFreeChart	0	4	18	0	0
Guava	177	61	190	43	21
Mockito	9	14	17	0	0
Spring Data JPA	0	20	16	2	5
JavaParser	12	1407	740	0	49
RxJava	2	132	316	0	4
Jackson Core	0	6	31	30	0
Kubernetes Client/-Java	788	386	1,777	11	3
Apache Commons IO	0	8	13	0	1
ScribeJava	2	4	7	3	0
JGroups	36	197	180	1	4
All Projects	1,024	2,239	3,305	90	87

incorrect method modifiers.

- **Throws Exception Mismatch:** This category reflects the number of cases in which we were unable to produce all of the exceptions that were declared in the method declaration.

Across all projects, *Argument Type Mismatch* is the highest contributor among all mismatch categories. *Return Type Mismatch* is the second most common mismatch type. We have found a relatively low number of *Modifier Mismatch* and *Throws Exception Mismatch*. The *Kubernetes Client/Java* project has the highest number of instances in all five mismatch categories. The relatively higher number of resolved method invocations compared to other projects justifies the outcome.

In order to identify the root cause of these *Mismatch* failures, we also performed a manual investigation for most instances of *Mismatch* failures.

(1) **Mismatch Type:** Inaccurate resolution of Formal Type Parameter Argument

Number of Instances: 291

Project: JavaParser

Root Cause: For generic method declarations, we can have arguments that are formal type parameters. The type of argument will be determined based on the type of argument that is passed during invocation. However, there are scenarios where the type of inferred argument

can be influenced by the outer method invocation. Listing 7 shows such a scenario. The method invocation that happened in line 14 is our topic of discussion. On line 23, from the method declaration context, we can see that the argument is a formal type parameter. The type of the passed argument on line 11 is *CommentMetaModel*. However, the outer method invocation on line 11 takes *Optional<BaseNodeMetaModel>* as the argument, and *BaseNodeMetaModel* is a superclass of *CommentMetaModel*. Hence, the compiler identifies the type of argument as *BaseNodeMetaModel*. Currently, our methodology does not take into account the context of an outer method invocation during the resolution of a formal type parameter.

Possible Solution: The consideration of any outer method invocation during the resolution of formal type parameter method argument will help us identify the accurate type of argument.

(2) **Mismatch Type:** Inaccurate resolution of Formal Type Parameter Method Return

Number of Instances: 222

Project: JavaParser

Root Cause: When we extend a generic class, we can provide type arguments during parent class declaration. The passed type arguments will be used to resolve the formal type parameters defined in the method declaration. Listing 8 demonstrates such a scenario. The method call *accept* happened on line 11 is our topic of discussion. Our approach failed to resolve the return type of the method invocation. The return type of the method is determined from the type arguments of the first method argument. The first argument of that method is an instance of *ThisExpression* which indicates *HashCodeVisitor* class as an argument. On the other hand, from the method declaration context, we can see *accept* method declaration takes *GenericVisitor* as the argument. Additionally, *GenericVisitor* is implemented by *HashCodeVisitor*, and type arguments are passed during class declaration. Hence, the compiler considers the type argument passed during class declaration for resolving formal type parameters. Currently, our approach does not support the consideration of passing type argument for non-generic argument type even though type can have generic class as parent class with type arguments in

Listing 7 Inaccurate resolution of Formal Type Parameter Argument (Excerpted from JavaParser)

```

// method invocation context
public final class JavaParserMetaModel {
    ...
    public static final CommentMetaModel commentMetaModel
        = new CommentMetaModel(Optional.of(nodeMetaModel));
    ...

    private static void initializePropertyMetaModels() {
        nodeMetaModel.commentPropertyMetaModel =
        new PropertyMetaModel(nodeMetaModel,
            "comment",
            com.github.javaparser.ast.comments.Comment.class,
            Optional.of(commentMetaModel),
            true, false, false, false);
    }
...
}

// method declaration context
public final class Optional<T> {
    ...
    public static <T> Optional<T> of(T value) {
        return new Optional<>(Objects.requireNonNull(value));
    }
    ...
}
```

the class definition.

Possible Solution: The consideration of a generic superclass for a non-generic class type and consideration of the type argument during the resolution of formal type parameters will resolve the issue.

(3) **Mismatch Type:** Inaccurate resolution of the class name of the method

Number of Instances: 64

Project: Kubernetes Client/Java

Root Cause: Listing 9 shows the method invocation *equals* on line 14 where our approach

Listing 8 Inaccurate resolution of Formal Type Parameter Method Return (Excerpted from JavaParser)

```
1
2  // method invocation context
3  public class HashCodeVisitor
4              implements GenericVisitor<Integer, Void> {
5      ...
6
7      public Integer visit(final BooleanLiteralExpr n,
8                           final Void arg) {
9          return (n.isValue() ? 1 : 0) * 31
10             + (n.getComment().isPresent()
11         ? n.getComment().get().accept(this, arg)
12         : 0);
13     }
14 ...
15 }
16
17 // method declaration context
18 public interface Visitable {
19     ...
20     <R, A> R accept(GenericVisitor<R, A> v, A arg);
21     ...
22 }
```

failed to identify the correct class name where the method is declared. Method declaration context shows the accurate class name of the method that is *V1ListMetaFluentImpl*. The class name of the method invoker is *V1ListMetaBuilder* [line 9] which is a sub-class of *V1ListMetaFluentImpl*. However, the byte-code representation also contains *equals* method for the sub-class *V1ListMetaBuilder*, and hence, our approach returns *V1ListMetaBuilder* as the class name of the method.

Possible Solution: There is no way to differentiate the methods added by the compiler in byte-code representation. Therefore, it will not be possible to resolve this issue using our current approach.

For *No Match* failures, we also performed a manual investigation of the root causes for most instances of *No Match* failures.

(1) **Mismatch Type:** Inaccurate resolution of the method invoker type

Listing 9 Inaccurate resolution of the class name of the method (Excerpted from Kubernetes Client/Java)

```java

// method invocation context
public class V1PersistentVolumeListFluentImpl
    <A extends V1PersistentVolumeListFluent<A>>
    extends BaseFluent<A>
    implements V1PersistentVolumeListFluent<A> {
    ...

    private V1ListMetaBuilder metadata;

    public boolean equals(Object o) {
        ...
        if (metadata != null
        ? !metadata.equals(that.metadata)
        : that.metadata != null) return false;
    }
...
}

// method declaration context
public class V1ListMetaFluentImpl
    <A extends V1ListMetaFluent<A>>
    extends BaseFluent<A>
    implements V1ListMetaFluent<A> {
    ...
    public boolean equals(Object o) {
    }
    ...
}
```

Number of Instances: 352

Project: Kubernetes Client/Java

Root Cause: Listing 10 shows the *ensureFieldAccessorsInitialized* method invocation on line 17, where our approach has failed to generate any method binding information. The root cause of this failure is the failure to identify the type of method invoker. In this example, the method invoker expression is a fully qualified name of the field instance declared on the class. However, our approach currently cannot resolve type from fully-qualified field expression.

Possible Solution: The appropriate resolution of fully-qualified field expression will resolve

this issue.

Listing 10 Inaccurate resolution of the method invoker type (Excerpted from Kubernetes Client/-Java)

```
// method invocation context
public final class V1alpha1Admission {
    ...
    private static final
    com.google.protobuf.GeneratedMessageV3.FieldAccessorTable
      internal_static_k8s_io_api_admission_v1alpha1
      _AdmissionReview_fieldAccessorTable;

    protected com.google.protobuf.GeneratedMessageV3
    .FieldAccessorTable
        internalGetFieldAccessorTable() {

        return io.kubernetes.client.proto.V1alpha1Admission
          .internal_static_k8s_io_api_admission_v1alpha1
          _AdmissionReview_fieldAccessorTable
          .ensureFieldAccessorsInitialized(
              io.kubernetes.client.proto.V1alpha1Admission
              .AdmissionReview.class,
              io.kubernetes.client.proto.V1alpha1Admission
              .AdmissionReview.Builder.class);
    }

...
}

// method declaration context
public abstract class GeneratedMessageV3 extends AbstractMessage
    implements Serializable {
    public static final class FieldAccessorTable {
        ...
        public FieldAccessorTable ensureFieldAccessorsInitialized(
            Class<? extends GeneratedMessageV3> messageClass,
            Class<? extends Builder> builderClass) {
        }
    ...
    }
}
```

(2) **Mismatch Type:** Inaccurate filtration based on method arguments comparison

Number of Instances: 41

Project: Jackson Core

Root Cause: Listing shows 11

shows the method invocation *System.arraycopy* on line 11, where our approach failed to generate any method binding information. Our approach failed to compare the arguments accurately in order to generate the method binding result. From the invocation context, we can see the first argument is a `char[]`, a primitive array, and from the declaration context, we can see the first argument is a *java.lang.Object*. Our approach currently does not support the inference of primitive array type to *java.lang.Object*.

Possible Solution: The possible solution to this issue is to consider the inference-based conversion from primitive array type to *java.lang.Object* during argument comparison.

(3) **Mismatch Type:** Inaccurate resolution of the method invoker type inside lambda expression

Number of Instances: 37

Project: JavaParser

Root Cause: Listing 12 shows a method invocation *report* on line 10 where our approach has failed to identify the appropriate type of invoker expression *reporter*. *reporter* is an argument of the lambda expression. The type of that argument will be determined from the enclosing *ClassInstanceCreation* type method invocation. Currently, our approach does not support the extraction of arguments when the enclosing node is an instance of *ClassInstanceCreation*.

Possible Solution: The possible solution to this issue is to resolve argument types for lambda expression arguments when they are enclosed inside *ClassInstanceCreation* node.

6.3 What is the execution time for resolving method binding information?

Motivation The evaluation of the performance of our strategy is an additional study subject that we wish to investigate. Our strategy can be viewed as two API calls. The initial API call loads all class meta-data from downloaded artifacts from Maven remote repositories. With the second API

Listing 11 Inaccurate filtration based on method arguments comparison (Excerpted from Jackson Core)

```java

// method invocation context
public class SerializedString
    implements SerializableString, java.io.Serializable
{
    ...
    public int appendQuoted(char[] buffer, int offset) {
        char[] result = _quotedChars;
        final int length = result.length;
        ...
        System.arraycopy(result, 0, buffer, offset, length);
    }

...
}

// method declaration context
public final class System {
    ...

    public static native void arraycopy(Object src,  int  srcPos,
                                        Object dest, int destPos,
                                        int length);
}
```

request, the method-binding information for any method-invocation node is resolved.

During the first API call, we store all the class meta-data from dependent artifacts if they aren't already recorded in our system. So, the response time of this API invocation is proportional to the amount of time needed to store the meta-data of all dependent artifacts. However, we will only need to perform this API call once per the whole project.

Thus, evaluating the execution time of our second API call is critical for the viability of our approach. Lower execution times for each method invocation will enhance the usefulness of our approach.

Result: Table 6.5 shows the execution time distribution among all evaluation projects. Among all evaluation projects, *Apache Commons IO* project has the least average execution time (99.37 ms)

Listing 12 Inaccurate resolution of the method invoker type inside lambda expression (Excerpted from JavaParser)

```

// method invocation context
public class Java8Validator extends Java7Validator {
    ...
    final Validator defaultMethodsInInterface =
    new SingleNodeTypeValidator<>
    (ClassOrInterfaceDeclaration.class,
            (n, reporter) -> {
                ....
                reporter.report(m,
                "'default' methods must have a body.");
            }
    );
...
}

// method declaration context
public class ProblemReporter {
    ...

    public void report(NodeWithTokenRange<?> node,
        String message, Object... args) {
    }
}
```

and *Kubernetes Client/Java* project has the largest average execution time (614.64 ms). In terms of median execution time, *Spring Data JPA* has the highest value with 221 ms, and *Apache Commons IO* project has the lowest value with 86 ms. Figure 6.1 displays the box plot diagram of the execution time for all evaluation projects.

Correlation between Number of Resolved Method and Execution Time: In table 6.5 we can see few projects (e.g., Kubernetes Client/Java) has relatively long execution time and also maximum execution time for few projects (i.e., JGroups) can take up to 17 seconds. To investigate the increased execution time, we performed another analysis to identify any correlation between the number of resolved methods and execution time. In order to resolve the method binding information of any method invocation, we may need to resolve other method invocations as well. One such scenario is that any argument of the method invocation itself is a method invocation expression.

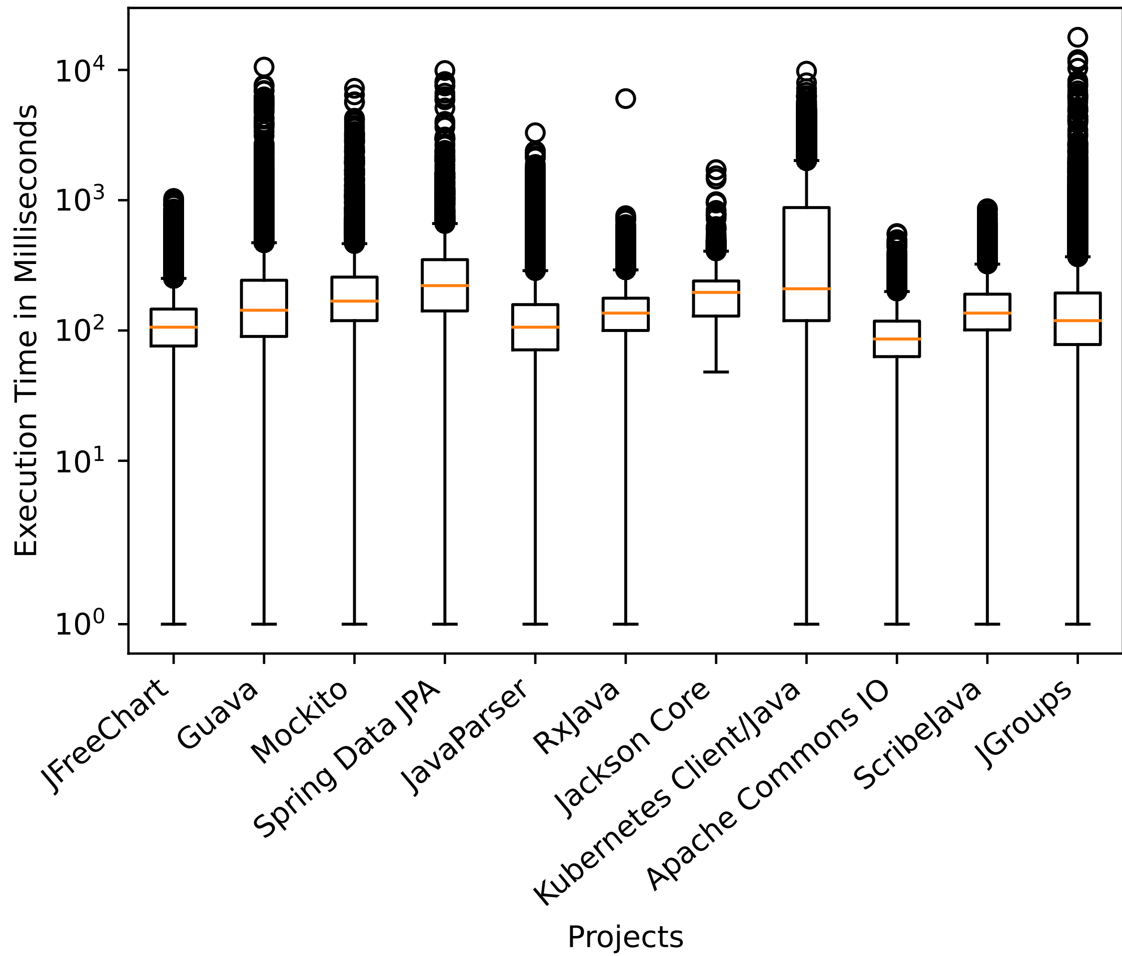


Figure 6.1: Box-plot of Execution Time for Projects

Table 6.5: Execution Time

Project Name	Execution Time in Milliseconds			
	Mean	Median	Min	Max
JFreeChart	134.06	106	1	1,032
Guava	221.79	143	1	10,518
Mockito	262.07	168	1	7,227
Spring Data JPA	348.27	221	1	9,900
JavaParser	148.46	106	1	3,295
RxJava	147.86	136	1	6,024
Jackson Core	190.02	196	48	1,717
Kubernetes Client/Java	614.64	209	1	9,783
Apache Commons IO	99.37	86	1	555
ScribeJava	166.34	136	1	860
JGroups	192.14	119	1	17,811
All Projects	380.93	150	1	17,811

Table 6.6: Distribution of execution time for Number of Resolved Methods

Number of Resolved Methods	Number of Data	Median Execution Time (Milliseconds)
1	206,088	136
2	51,728	184
3	7,815	172
4	5,817	190
5-44	1,904	301

Hence, we need to resolve the method binding information of that method invocation in order to determine the type of the argument. For our analysis, we have grouped execution time across all evaluation projects against the total resolution of method-binding results for each invocation. Table 6.6 displays the number of resolved methods and their corresponding average execution time. Due to the rarity of a large number of method resolutions per invocation, we have grouped all the resolved method numbers from 5 to 44 into one group and considered it as one single group. Figure 6.2 shows the correlation between execution time and the total number of resolved methods per invocation, where we will see an increasing number of execution times with the increase of internal method resolutions.

Execution Time Comparison Between External And Internal Method Invocation: We performed execution time analysis between external and internal method invocation. We consider

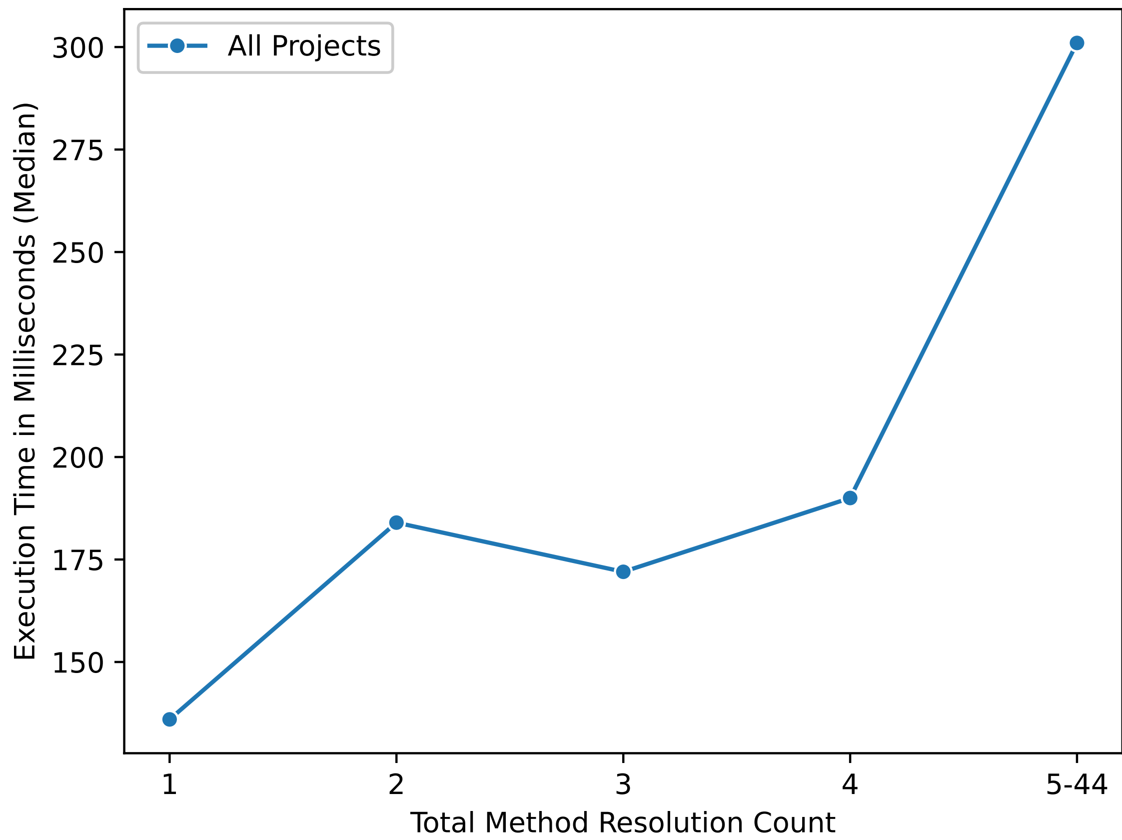
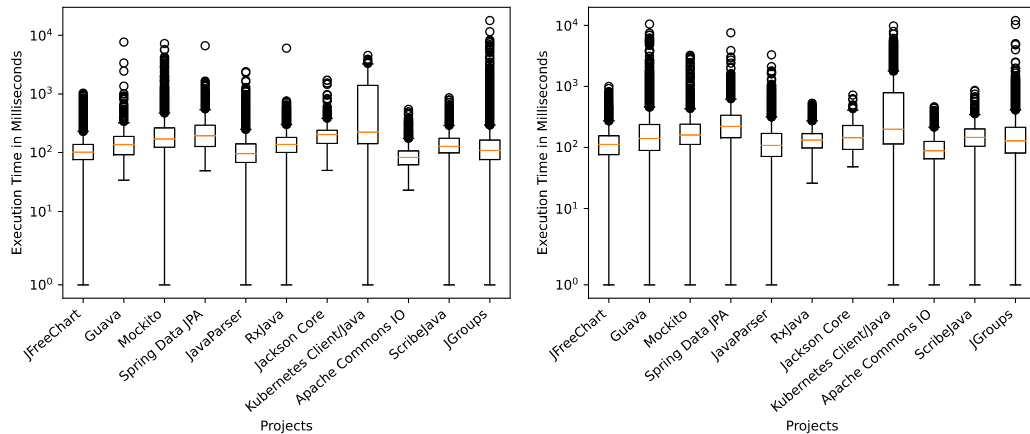


Figure 6.2: Median of execution Time over Resolved Method Count



(a) Internal Method Invocation (b) External Method Invocation

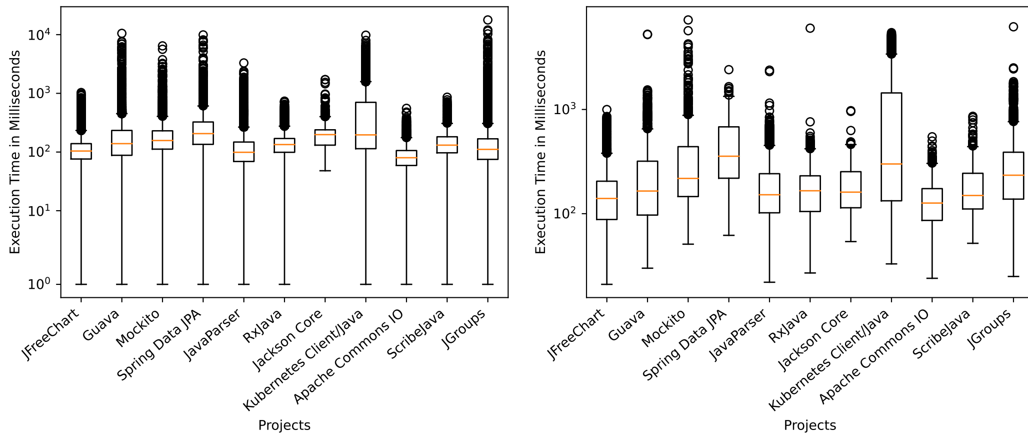
Figure 6.3: Execution Time for Internal and External Method Invocation

internal method invocation if that particular method is declared in any class of the project. On the other hand, we consider any method invocation as external if that particular method invocation is declared in any dependent external libraries or declared in Java core package classes. Figure 6.3 shows the box-plot diagram of both internal and external method invocations. From the plot, we can see nearly identical execution times for both categories of method invocations. From the figure, we can come to the conclusion that both external and internal invocation will have a similar execution time.

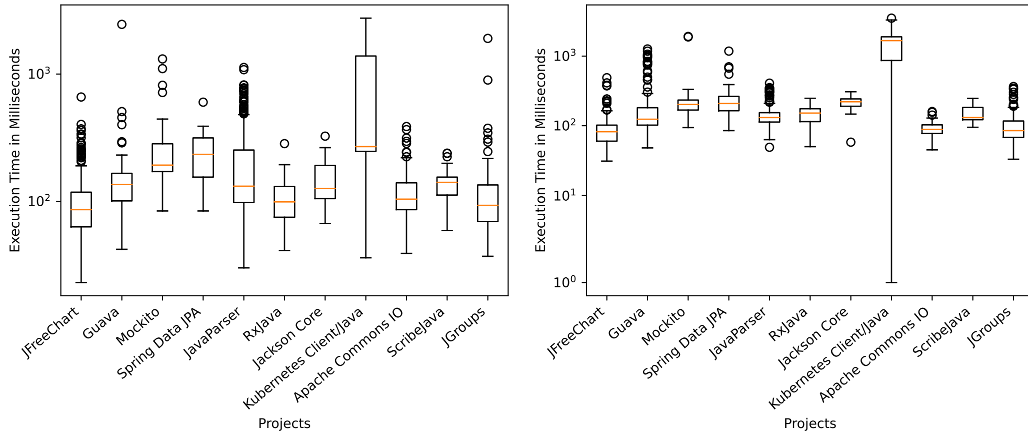
Execution Time Comparison Among Different Method Invocation Expressions: We have performed another analysis where we investigated execution time based on method invocation expression types. Figure 6.4 shows the execution time box plot for all types of method invocation expressions. Since method invocation expression is most prevalent among all invocation expression types, the execution time box plot of method invocation expression reflects the overall execution time distribution across all projects.

6.4 Comparison between "API-Finder" and "JavaSymbolSolver"

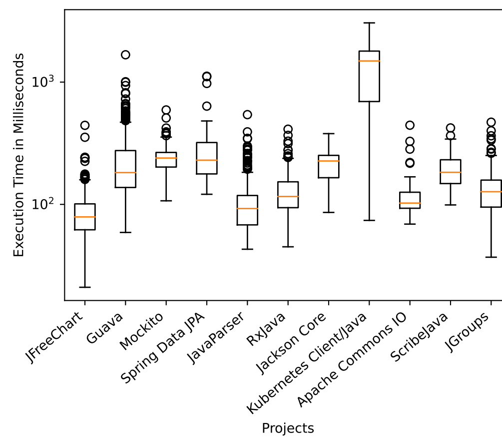
Among all existing tools, JavaSymbolSolver can identify method binding information with more precision. We have conducted a limited evaluation to identify the accuracy of both JavaSymbolSolver and API-Finder with Eclipse JDT Compiler. There are several limitations that restricted us



(a) Method Invocation (b) Class Instance Creation



(c) Constructor Invocation (d) Super Method Invocation



(e) Super Constructor Invocation

Figure 6.4: Execution Time for Method Invocation Expressions

Table 6.7: Accuracy comparison between API-Finder and JavaSymbolSolver

Project Name	JavaSymbolSolver			API Finder		
	Success	Failure	Accuracy	Success	Failure	Accuracy
JFreeChart	29,939	1,577	95%	31,445	71	99.77%
Guava	20,224	6,000	77.12%	25,530	694	97.33%
Apache Commons IO	2,572	1,331	65.89%	3,839	64	98.36%

to perform the complete comparison of API-Finder with JavaSymbolSolver. The limitations are:

- JavaSymbolSolver cannot extract all dependent artifact archives from the project. JavaSymbolSolver assumes that we have all the required archives and requires the archives to be passed as arguments for the resolution of method binding information. For this evaluation, we needed to provide all the dependent artifacts archives with manual intervention. For this comparison, we have selected three projects (i.e., JFreeChart, Guava, Apache Commons IO) with minimal dependent artifacts.

- JavaSymbolSolver depends on JavaParser, an AST parsing library. JavaParser does not have a separate representation for *ConstructorInvocation*, *SuperMethodInvocation*, or *SuperConstructorInvocation*. Therefore, we have limited our evaluations for *MethodInvocation* and *ClassInstanceCreation* type method invocation instances.

Table 6.7 shows the accuracy comparisons for 3 evaluation projects. In all three projects, API-Finder has outperformed JavaSymbolSolver in terms of accuracy. For projects with a higher percentage of generics and lambda expression or method references (i.e., Guava, Apache Commons IO), API-Finder has outperformed significantly.

6.5 Limitations and Threats to Validity

Our proposed approach has a few limitations which can impact the resolution of accurate method-binding information.

- **Dependency on the Build System:** Our approach collects dependent artifacts information and java version from build files. Currently, we only support two build systems (i.e., Gradle,

and Maven). Therefore, we would be unable to retrieve dependent artifact information or the Java version for projects with an unrecognized build system or no build system.

- **Availability of dependent Artifacts in Public Maven Repository:** The extracted artifacts from the project has to be available in Maven public artifact repository to download the JAR and extract the class meta-data. We would be unable to process artifacts if they are hosted in a private artifact repository. In addition, if the artifact version is not available in the public artifact repository, we would be unable to process the artifact and extract appropriate information which will have an impact during the resolution of method binding information.

- **Inability to identify method-binding From Test Packages:** We collect all class information from the project's archivable artifacts. Test packages are not usually available in the released artifacts. Therefore, our approach cannot identify any method-binding information if the method is declared on a test package.

- **Remote Fetching of Gradle Build related files** Gradle build scripts can be written with *Groovy* or *Kotlin* language. For the extraction of build-related files, we perform a regex-based approach to search all the build-related files in the remote fetch mechanism. Since the build system can be complex and identification of all build-related files from the build script can be challenging. Therefore, due to the limitation of the regex-based approach, we would not be able to successfully fetch all related files and perform the extraction of artifacts and Java version for the complex build process.

Internal Validity In our experiment we have compared the produced result of our approach with a compiler implementation. The compiler will be able to generate accurate method-binding information when the complete codebase is available. Therefore, we can ensure the credibility of our accuracy rate. Additionally, the compiler can also produce more detailed information for each method binding information. For example, for a parameterized type argument compiler store the complete type information of the type parameters. For the comparison of the return type or argument types in the evaluation we have only considered the qualified class name. Therefore, the type parameters of any parameterized argument or return type generated by our approach can have

differences. However, there are other scenarios where the identification of the appropriate type representation of the type parameter will lead to the identification of the qualified class name of the argument or return types. Therefore, we have confidence in the identification of a complete type representation of return and argument types.

External Validity We have performed our experiment on a relatively small number of open-source projects. However, we have tried to select from diverse backgrounds and we have witnessed a consistent rate of accuracy among all our evaluation projects. Therefore, we have confidence that our accuracy and execution time can be replicated in other projects as well.

Chapter 7

Conclusion and Future Work

In this thesis, we have proposed a streamlined approach where we start by providing support for extracting artifacts and the project's Java version, and finally provide an independent systematic strategy that is independent of any existing AST implementation to identify accurate method binding information from invocation context. We evaluated our approach against a compiler implementation (i.e., Eclipse JDT) for eleven popular open-source projects in order to measure the accuracy of our approach.

Our result shows that we have identified accurate method-binding information with an accuracy rate above 93%, where JFreeChart has the highest success rate with an accuracy of 99.78% and the JavaParser has the lowest success rate with an accuracy of 93.31%. We also evaluated the execution time for resolving method-binding information. Our result shows that, on average, each resolution of method-binding information takes 380.93 ms.

During the evaluation of execution time, we also discovered additional insights.

- There is no significant variation in the distribution of execution time for the project's internal and external dependencies. Our approach will have a similar execution time for both types of method-binding resolutions.

- There is a positive correlation between the number of internal method-binding resolutions during a resolution of method-binding information and execution time. Extracting method-binding information for a method invocation expression may lead to other method-binding

resolutions. The total execution time will increase with the number of internal method-binding resolutions.

7.1 Applications of "API Finder"

API Finder can be utilized in diverse scenarios. We can utilize "API Finder" to identify method binding information from the diverse origins where we can only access partial programs such as:

- We can show method binding information for source code available on the GitHub Platform.
- We can show method meta-data for the partial fragments of codes that are available in Stack-overflow[1] platform, or for API usage code snippets available in websites, such as Android[2].
- We can integrate "API Finder" with other web-based services such as ChatGPT[3] generated codes.

7.2 Future Work

In the future, we plan to accommodate the language support for the newer Java versions. We also plan to provide a more robust and efficient approach to extracting dependent artifact information extraction from build system files. Additionally, we want to explore the research direction of identifying method-binding information from code fragments that are available in forums and websites using our approach.

[1] https://stackoverflow.com/
[2] https://developer.android.com/docs
[3] https://openai.com/blog/chatgpt

Bibliography

[1] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using findbugs on production software," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 805–806. [Online]. Available: https://doi.org/10.1145/1297846.1297897

[2] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a static noninteractive approach to feature location," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 2, p. 195–226, apr 2006. [Online]. Available: https://doi.org/10.1145/1131421.1131424

[3] S. Mani, R. Padhye, and V. S. Sinha, "Mining api expertise profiles with partial program analysis," in *Proceedings of the 9th India Software Engineering Conference*, ser. ISEC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 109–118. [Online]. Available: https://doi.org/10.1145/2856636.2856646

[4] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, ser. OOPSLA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 313–328. [Online]. Available: https://doi.org/10.1145/1449764.1449790

[5] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 802–811.

[6] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[7] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 307–318.

[8] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: A search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 681–682. [Online]. Available: https://doi.org/10.1145/1176617.1176671

[9] T. Clem and P. Thomson, "Static analysis at github: An experience report," *Queue*, vol. 19, no. 4, p. 42–67, sep 2021. [Online]. Available: https://doi.org/10.1145/3487019.3487022

[10] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 383–392. [Online]. Available: https://doi.org/10.1145/1595696.1595767

[11] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017, e1838 smr.1838. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1838

[12] H. Zhong and X. Wang, "Boosting complete-code tool for partial program," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17. IEEE Press, 2017, p. 671–681.

[13] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proceedings of the Twenty-Second IEEE/ACM International*

Conference on Automated Software Engineering, ser. ASE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 204–213. [Online]. Available: https://doi.org/10.1145/1321631.1321663

[14] E. M. Gagnon, L. J. Hendren, and G. Marceau, "Efficient inference of static types for java bytecode," in *Static Analysis*, J. Palsberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 199–219.

[15] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, p. 13.

[16] L. Gasparini, E. Fregnan, L. Braz, T. Baum, and A. Bacchelli, "Changeviz: Enhancing the github pull request interface with method call information," in *2021 Working Conference on Software Visualization (VISSOFT)*, 2021, pp. 115–119.

[17] D. Dam. diptopol/apifinder. [Online]. Available: https://github.com/diptopol/apifinder

[18] T. E. Foundation. Eclipse java development tools (jdt). [Online]. Available: https://www.eclipse.org/jdt/