



AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

DZIEDZINA: Nauk inżynieryjno-technicznych

DYSCYPLINA: Informatyka techniczna i telekomunikacja

ROZPRAWA DOKTORSKA

Opracowanie i weryfikacja nowych metod analizy struktury
i sposobu działania złożonych systemów informatycznych

Autor: Krzysztof Borowski

Promotor rozprawy: dr hab. inż. Bartosz Baliś, AGH

Promotor pomocniczy: dr inż. Tomasz Stanisław Orzechowski, VirtusLab

Praca wykonana:

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Informatyki, Elektroniki i Telekomunikacji

Instytut Informatyki

Kraków, 2023



FIELD OF SCIENCE: Engineering and technology

SCIENTIFIC DISCIPLINE: Information and communication technology

DOCTORAL THESIS

Development and verification of new methods for analyzing
the structure and operation of complex IT systems

Author: Krzysztof Borowski

First Supervisor: dr hab. inż. Bartosz Baliś, AGH

Assisting Supervisor: dr inż. Tomasz Stanisław Orzechowski, VirtusLab

Completed in:

AGH University of Krakow

Faculty of Computer Science, Electronics and Telecommunications

Institute of Computer Science

Kraków, 2023

I would like to express my gratitude to dr Bartosz Baliś for his kindness, dedication, time, valuable ideas and professional guidance throughout whole research; to VirtusLab R&D department for the implementation effort in improving created solutions and in particular to Rafał Pokrywka and dr Tomasz Orzechowski for continuous support; to my dear longtime Friends for all the help on this journey; and, last but not least, to my parents Jan and Beata, my brothers, and to my wife Dorota for motivation and always believing in me. With dedication to Dorota and my newborn son Tomasz.

Praca powstała w ramach programu "Doktorat Wdrożeniowy" finansowanego przez Ministerstwo Nauki i Szkolnictwa Wyższego. Badania zostały przeprowadzone we współpracy z firmą VirtusLab.

—

The research presented in this dissertation was partially supported by the "Doktorat Wdrożeniowy" program of the Polish Ministry of Science and Higher Education. The research was conducted in cooperation with VirtusLab.

Streszczenie

Zrozumienie oprogramowania może być niezwykle czasochłonne ze względu na stale rosnący rozmiar kodu źródłowego programu. W związku z tym istnieje rosnąca potrzeba przyspieszenia procesu rozumienia kodu w celu ułatwienia utrzymania i zmniejszenia związanych z tym kosztów. Kluczowym aspektem tego procesu jest zrozumienie i zachowanie wysokiej jakości struktury zależności kodu. Chociaż istnieje już wiele różnych reprezentacji struktury i zależności kodu, zaskakujący jest brak modeli, które ściśle reprezentują kod źródłowy i koncentrują się na wspieraniu zrozumienia oprogramowania. W rezultacie nie istnieją łatwo dostępne i gotowe do użycia narzędzia pomagające w zrozumieniu zależności, ich refaktoryzacji i monitorowaniu ich jakości. Aby wypełnić tę lukę, proponujemy Semantic Code Graph (SCG), model informacyjny, który oferuje szczegółową abstrakcyjną reprezentację zależności kodu z możliwie dokładnym odwzorowaniem kodu źródłowego. W niniejszej pracy określamy krytyczne właściwości modelu SCG i demonstrujemy jego implementację dla języków Java i Scala. Aby zweryfikować przydatność modelu SCG w zrozumieniu oprogramowania, porównujemy go z dziewięcioma innymi modelami reprezentacji kodu źródłowego. Ponadto wybieramy 11 znanych oraz szeroko stosowanych projektów z otwartym kodem źródłowym napisanych w Javie i Scali i wykonujemy na nich szereg czynności związanych ze zrozumieniem oprogramowania przy użyciu trzech różnych modeli reprezentacji kodu: proponowanego SCG, Call Graph i Class Collaboration Network. Następnie analizujemy jakościowo wyniki, aby porównać wydajność tych modeli pod względem możliwości wspierania rozumienia oprogramowania. Działania te obejmują interaktywną wizualizację zależności kodu, zrozumienie struktury projektu, identyfikację krytycznych elementów projektu, odzyskiwanie architektury oprogramowania poprzez partycjonowanie oprogramowania oraz odkrywanie podobieństw kodu poprzez zaawansowaną eksplorację grafowego modelu reprezentacji kodu. Nasze prace udowadniają, że SCG zwiększa możliwości rozumienia oprogramowania w porównaniu z dominującymi modelami Class Collaboration Network i Call Graph. Co więcej, analiza danych oparta na SCG dostarcza gotowych do wdrożenia spostrzeżeń na temat poprawy czytelności oprogramowania. Udostępniamy również narzędzie open-source, *scg-cli*, aby pomóc w odtworzeniu wyników i dalszych badaniach. Wierzimy, że opisana praca jest krokiem w kierunku następnej generacji narzędzi, które usprawnią rozumienie i zarządzanie zależnościami kodu.

Abstract

Software comprehension can be extremely time-consuming due to the ever-growing size of codebases. Consequently, there is an increasing need to accelerate the code comprehension process to facilitate maintenance and reduce associated costs. A crucial aspect of this process is understanding and preserving the high quality of the code dependency structure. While a variety of code structure models already exist, there is a surprising lack of models that closely represent the source code and focus on software comprehension. As a result, there are no readily available and easy-to-use tools to assist with dependency comprehension, refactoring, and quality monitoring of code. To address this gap, we propose the Semantic Code Graph (SCG), an information model that offers a detailed abstract representation of code dependencies with a close relationship to the source code. We establish the critical properties of the SCG model and demonstrate its implementation for Java and Scala languages. To validate the SCG model's usefulness in software comprehension, we compare it to nine other source code representation models. Additionally, we select 11 well-known and widely-used open-source projects developed in Java and Scala and perform a range of software comprehension activities on them using three different code representation models: the proposed SCG, the Call Graph, and the Class Collaboration Network. We then qualitatively analyze the results to compare the performance of these models in terms of software comprehension capabilities. These activities encompass interactive visualization of code dependencies, project structure comprehension, identifying critical project entities, recovering software architecture through software partitioning and uncovering code similarities through software mining. Our findings demonstrate that the SCG enhances software comprehension capabilities compared to the prevailing Class Collaboration Network and Call Graph models. Moreover, the SCG-based data analysis yields actionable software comprehension insights. We also release an open-source tool, *scg-cli*, to assist with result reproduction and further research. We believe that the work described is a step towards the next generation of tools that streamline code dependency comprehension and management.

Contents

List of Figures	xiii
1 Introduction	1
1.1 Research motivation	1
1.2 Research objectives and methodology	2
1.3 Scientific contribution	3
1.4 Publications	3
1.5 Dissertation organization	4
2 Related work	5
2.1 Code dependency representations	5
2.2 Tools for extracting code dependency structure	7
2.3 Software comprehension tools	9
3 Semantic Code Graph	11
3.1 Source code dependency model requirements	11
3.2 Semantic Code Graph base model	12
3.3 Semantic Code Graph for Scala and Java	13
3.4 Semantic Code Graph extraction process	15
3.4.1 Implementation overview	15
3.4.2 Extracting SCG for the Scala2 language	17
3.4.3 Extracting SCG for the Scala3 language	17
3.4.4 Extracting SCG for the Java language	17
3.4.5 Challenges in extracting SCG from Bytecode	18
3.4.6 Stable symbol identifiers for a multi-language SCG model	21
3.4.7 SCG storage format	21
4 Outline of research objectives and methods	25
4.1 Research objective	25
4.2 Selection of models	25
4.3 Data collection	26
4.4 Tools	27
4.5 Empirical evaluation	27

5	Software Comprehension capabilities: comparison of SCG and other graph-based code representations	29
5.1	Introduction	29
5.2	General model capabilities	29
5.3	Support for software comprehension use cases	31
5.4	Tool adoption readiness	32
5.5	Summary	33
6	Code dependency comprehension through interactive visualization – <i>Graph Buddy</i> IDE plugin	35
6.1	Motivation and significance	35
6.2	Related work	36
6.3	Graph Buddy information model	36
6.4	Graph Buddy IDE’s plugins	37
6.4.1	Graph Buddy architecture	37
6.4.2	Graph Buddy user interface	38
6.5	Graph Buddy evaluation	40
6.5.1	Graph Buddy performance	40
6.5.2	Graph Buddy user feedback loop	40
6.5.3	Graph Buddy usability user survey	40
6.6	Conclusions and future work	44
7	<i>scg-cli</i> command line tool to facilitate software comprehension	45
7.1	Motivation and significance	45
7.2	Related work	46
7.3	Software description	47
7.3.1	Software architecture	47
7.3.2	Software functionalities	48
7.4	Illustrative Examples	51
7.4.1	Generating SCG data	51
7.4.2	Project overview	51
7.4.3	Finding crucial nodes	51
7.4.4	Project partitioning	54
7.4.5	Exporting and analyzing in external tools	55
7.5	Impact	56
7.6	Conclusions	57
8	Empirical software comprehension with Semantic Code Graph	59
8.1	Introduction	59
8.2	Project overview	60
8.3	Project structure comprehension	64
8.4	Software architecture recovery through software module clustering	70

8.5	Discovering critical project entities	81
8.6	Finding code similarities	87
9	Summary of results	91
9.1	Answering research questions	91
9.2	Impact of results	92
10	Conclusions	95
	Bibliography	97

List of Figures

3.1	SCG visualization for a simple inheritance example.	15
3.2	SCG visualization of method with local calls and local value τ definition.	15
3.3	Front-end analysis steps in a compiler	16
3.4	Scala 2 compiler phases. All type related information are available after the <i>typer</i> phase. . .	17
3.5	Scala 3 compiler phases, with <i>pickler</i> phase for storing TASTy binary files.	18
3.6	Java bytecode does not contain the source code location of non-runtime code elements even with the <code>javac -g</code> flag. For other elements only the line number can be recorded.	19
3.7	Java source code: The <i>Integer</i> type for the generic <i>TypeErasure</i> class will not be represented in bytecode for the local variable <i>a</i>	20
3.8	The Scala compiler generates the <i>B#foo()</i> method with reference to the generated static method <i>A#foo\$(a)</i>	20
3.9	Comparison of a call graph generated from bytecode and from the source code – <i>B#foo()</i> and <i>A#foo\$(a)</i> methods do not exist in the source code.	21
6.1	Graph Buddy loading and introduction screen.	38
6.2	Graph Buddy architecture – SCG protobuf files are the source information model for visualization.	38
6.3	Graph Buddy user interface with partial graph visualization of MetalsLanguageServer class – its two declared methods, <i>cancelAll</i> directly calling <i>cancel</i> , and method <i>cancel</i> overriding method defined in <i>Cancellable</i> trait extended by <i>MetalsLanguageServer</i> . The user can interactively inspect the related source code by selecting the node or edge, analyze additional dependencies through a context menu, or filter nodes or edges by their type. . . .	39
6.4	The dependencies of available code actions in the <i>metals</i> library presented in Graph Buddy .	41
6.5	Graph Buddy (left) vs IntelliJ’s call graph solution to task T1	42
6.6	Graph Buddy (left) vs IntelliJ find a path solution to task T2 . The text is highlighted artificially to mark the possible call path between the <i>rename</i> and <i>isVarSetter</i> methods.	43
7.1	The <i>scg-cli</i> software architecture. The <i>load/generate data module</i> generates SCG data files for a project. The <i>project analysis module</i> performs analysis of the project using the SCG data. The <i>export module</i> enables custom analysis using external tools known to researchers, such as Jupyter.	48
7.2	Project overview for the <i>commons-io</i> project generated with the scg-cli summary command.	52

7.3	Top nodes for the <i>commons-io</i> project. Colors denote partitions (modules) obtained from 6-partition split using the <i>gpmetis</i> algorithm. Visualized with Gephi.	56
8.1	Class Collaboration Network example with <i>INHERITANCE</i> dependencies between <i>class B</i> and <i>class A</i> , <i>AGGREGATION</i> representing variable <i>D</i> and <i>REFERENCE</i> representing method param and return type coupling.	60
8.2	Top 6 SCG nodes distribution in analyzed projects.	61
8.3	Top 5 SCG edges distribution in analyzed projects.	62
8.4	Nodes' incoming and outgoing degree distribution for SCG model extracted for each analyzed project.	67
8.5	The incoming and outgoing degree distribution of nodes in the SCG, CCN, and CG models was extracted for the <i>retrofit</i> project, with a focus on identifying outlier nodes with high outgoing degrees in the SCG and CG networks.	68
8.6	The incoming and outgoing degree distribution of nodes in the SCG, CCN, and CG graphs extracted for the <i>akka</i> project. Nodes with high incoming degrees, such as the <i>ActorRef</i> class and the method <i>ActorRef#!</i> , serve as the primary building blocks for the <i>akka</i> project.	68
8.7	CCN visualization and degree distribution for the RxJava project confirming high tendency for global clustering.	69
8.8	Visualization [40] of the <i>metals</i> project SCG graph, for 2-5 partitions, using <i>gpmetis</i> with high partition size imbalance allowed. High-degree nodes are labeled.	76
8.9	CCN visualization [40] of the <i>metals</i> project, for 2-5 partitions, using <i>gpmetis</i> with high partition size imbalance allowed. High-degree nodes are labeled.	77
8.10	CCN visualization [40] of four projects: <i>commons-io</i> , <i>RxJava</i> , <i>akka</i> , <i>spark</i> . Colors denote different partitions computed with <i>gpmetis</i> , with maximum partition imbalance allowed. For all projects we can visually localize clearly separated parts and no single partition that dominates the entire project. High-degree nodes are labeled.	79
8.11	CG graph visualization [51] of the <i>metals</i> project, 2-5 partitions, using <i>gpmetis</i> with high partition size imbalance allowed. Nodes with a high Betweenness Centrality are labeled.	80
8.12	Method <i>oldReloadResult</i> and <i>oldReloadResult</i> from <i>metals</i> project are similar, differing only in naming and the first condition statement. Consequently, a <i>common</i> method shown in Listing 8.9 can be used to replace them.	89

Chapter 1

Introduction

1.1 Research motivation

Software is becoming increasingly complex and larger, driven by fast-changing business requirements and advances in programming languages, libraries, and runtime infrastructure. At the same time, the quality of software architecture, especially in such a dynamic environment, deteriorates over time [34], so programmers need to put growing amount of effort into understanding the source code [2] in order to make required changes or introduce new features important from a product or service perspective. This “activity of understanding existing software systems” is called *software comprehension* or *code comprehension* [85].

Multiple measures are applied to facilitate software comprehension by both keeping the code clean and speeding up the code-understanding process. Performing code reviews [73], following a written set of conventions [108], measuring various software quality metrics [55], or linting the source code [92] (e.g. guarding against common anti-patterns) have a positive impact on written code quality. Additionally, Integrated Development Environments (IDEs) help programmers browse, read, comprehend, and maintain code efficiently through common IDE features such as syntax highlighting, code navigation, or advanced refactoring capabilities. Nevertheless, Xia et al. [123] argue that up to 58 percent of software maintenance activities are related to software comprehension [107, 111] and claim that we need more advanced software comprehension tools. With the ever greater importance of software and rapidly growing code bases, even small reduction of time spent on software related activities will bring significant cost savings.

One of the major code comprehension challenges is understanding and managing dependencies of software entities. *Understanding* code dependencies may comprise finding key code entities [68], localizing connected elements [2, 3], answering different reachability questions [64], reasoning about entire software structure including software architecture understanding [37, 69, 102]. *Managing* code dependencies denotes introducing required bug fixes or new features and, at the same time, guarding the project structure against decay signs, such as high coupling between modules or classes [32], data abstraction coupling or message passing coupling [93], forming god objects [117], introducing unintended dependencies between code entities, etc. [12]. Understanding and managing code dependencies is less time consuming if the source code structure is of high quality. To maintain a high quality code dependency structure, we can either prevent the erosion process or reorganize the already affected code [84]. Preventing harmful dependencies requires

constant code dependency and structure monitoring which, although important from a project cost and maintainability perspective, is not something widely applied. Quality metrics most frequently used in projects, e.g., cyclomatic complexity [72] or Lines of Code, focus on a particular method or class and do not assess the quality of code dependency structure. Consequently, proper methods and tools are required to discover, understand, and improve the structure of existing code.

Regarding research on models for code dependency structure, there is a substantial body of work demonstrating the utility of software network analysis on various theoretical models [3, 8, 9, 41, 49, 52, 68, 70, 79, 89, 95, 101, 102, 115, 119]. However, even in the case of widely researched models, such as Call Graph (CG) [3, 46, 60, 64, 98] or Class Collaboration Network (CCN) [13, 31, 68, 119], the underlying implementations, developed by the researchers to extract and analyze these software models, are rarely published. Consequently, it is very difficult to reproduce the results or pursue further research in the code dependency comprehension field, especially leveraging new or commercial projects. Such a lack of high quality tools for extracting and analyzing the software dependency structure forces researchers to start from scratch every time. This not only hampers research and efforts, but also makes it impossible to effectively compare different results. Perhaps for this reason **we have found no research devoted to comparative empirical evaluation of different code representation models**. Furthermore, due to the low emphasis on the practical aspects of previously conducted research, there is little to no impact on software comprehension from the code dependency angle in the commercial software development process. We argue that this is mostly due to the **lack of high-quality and well-established stable abstract code dependency model focused on practical usability aspects** and, in consequence, tools leveraging such a model to speed up software comprehension and code refactoring process. Code dependency analysis should not only be confined to a theoretical level but, equally importantly, should be geared towards practical applications, reproduction of results, and further research.

1.2 Research objectives and methodology

To address the research gap between theoretical code dependency models and their applications, we present *Semantic Code Graph* (SCG), a code dependency information model which captures the structure and semantics of code dependencies in a software project while preserving the direct relation of the representation to the source code. The **main objective of our research is to evaluate the effectiveness of the SCG for software comprehension activities**. To this end, we formulate the following two research questions:

- **RQ1:** Does the SCG model enhance software comprehension capabilities in comparison with the Class Collaboration Network (CCN) and the Call Graph (CG) models?
- **RQ2:** Does SCG-based data analysis enable actionable software comprehension insights?

We conduct the research addressing these questions using the following steps. First, in order to assess the theoretical suitability of the SCG model in software comprehension, we compare it to nine other source code representation models. Then, we prove SCG suitability for detailed interactive code dependency visualization through creating *Graph Buddy* IDE plugin solution and its evaluation. Additionally, we design,

implement and publish *scg-cli* command line tool, enabling various software comprehension activities over Java or Scala projects based on extracted SCG model. Using *scg-cli* tool, we conduct an empirical study involving eleven popular open source projects. We utilize these models to comprehend project structures, identify critical project entities along with their dependencies, and uncover code similarities. We demonstrate that both CCN and CG can be effectively extracted from the corresponding SCG model. Furthermore, while each of these models provides a distinct perspective on the project, the SCG stands out as the most comprehensive, enabling analyses that surpass the capabilities of the other two.

1.3 Scientific contribution

The contributions of this dissertation can be summarized as follows:

- We define and describe the abstract Semantic Code Graph model and its concrete representations for Java and Scala languages. SCG for Java and Scala is able to describe various types of dependencies between software entities at different levels – from classes and methods, down to local value definitions and type declarations.
- We define and implement SCG intermediate representation and storage format to facilitate code dependency data analysis and visualizations with external tools and to support effective model extraction for new languages.
- We design and implement robust interactive code dependency visualization based on SCG model. We publish the solution as *Graph Buddy*¹ plugin for IntelliJ and VSCode IDEs and conduct *Graph Buddy* usefulness evaluation.
- We conduct an empirical analysis of the SCG model’s performance concerning its software comprehension capabilities in comparison to the CCN and CG, and qualitatively analyze its results.
- To enable reproducibility and further research, we publish tools and data used in our research. These include the extracted SCG data for the eleven open-source projects used in the empirical study, and the open-source tool *scg-cli*². The tool is capable of extracting the SCG data for Java projects. It also supports SCG-based data analysis capabilities, and exporting SCG data for usage in external analysis tools, such as *Jupyter Notebook* or *Gephi*.

1.4 Publications

The following papers were produced from research within this thesis:

- Krzysztof Borowski, Bartosz Baliś, and Tomasz Orzechowski. Graph buddy — an interactive code dependency browsing and visualization tool. In *2022 Working Conference on Software Visualization (VISsOFT)*, 2022 [19]

¹<https://github.com/VirtusLab/graphbuddy>

²<https://github.com/VirtusLab/scg-cli>

- Krzysztof Borowski, Bartosz Baliś, and Tomasz Orzechowski. Semantic code graph – an information model to facilitate software comprehension, arXiv:2310.02128 [cs.SE], 2023 [21]
- Krzysztof Borowski and Bartosz Baliś. scg-cli – a tool supporting software comprehension via extraction and analysis of semantic code graph, arXiv:2310.03044 [cs.SE], 2023 [20]

Because Chapters 6 and 7 are based on publications, to some extent they have a self-contained structure resembling a research paper, e.g. with their own *Related work* and *Evaluation* sections.

1.5 Dissertation organization

This dissertation is structured as follows. Chapter 2 presents related work. In Chapter 3, we introduce the SCG model and its detailed implementation for Java and Scala. We also discuss the extraction process of SCG data and propose an intermediate storage format. Chapter 4, serves as an introduction to subsequent evaluation chapters, outlining the research objectives and methods applied to achieve them. In Chapter 5, we evaluate software comprehension capabilities of the SCG and nine other established graph software representations. Chapter 6 describes a practical application of the SCG for interactive software visualization. Chapter 7 introduces the *scg-cli* command line tool facilitating SCG-based software comprehension activities. Chapter 8 contains a comprehensive empirical evaluation of the proposed SCG model by demonstrating diverse software comprehension activities conducted on eleven open-source projects. Chapter 9 provides answers to research questions and discusses the impact of research results. Chapter 10 concludes the dissertation.

Chapter 2

Related work

In this section we review related work, first focusing on existing code dependency representations, then on tools capable of extracting and analyzing various code dependency models, and finally on software comprehension tools. We restrict the review to tools that provide support for either Java or Scala.

2.1 Code dependency representations

Early graph representations of programs were focused on program execution and dependencies between code segments. One of the first commonly analyzed code graph models is Control Flow Graph (CFG) [7] – a directed graph where nodes are basic program blocks and edges represent control flow paths. CFG allows determining the exact execution order of a program including different program paths implied by control conditions (like if statements). CFG can be successfully applied to program analysis, such as finding unreachable code, compiler optimizations, code generation, code debugging and others [80]. CFG focuses on block’s execution paths and does not represent dependencies on the statement level. To that end, Ferrante et al. [38] introduced the concept of Program Dependence Graph (PDG) which focuses on single statement dependencies. In PDG code statements are represented as nodes, and edges express two main relations between them:

- data dependency (execution of one statement requires data produced by another statement),
- control flow (execution of one statement depends on a control condition evaluated by another statement).

Control dependencies in PDG are commonly extracted from the Control Flow graph but represent a dependency tree of statements rather than an exact execution path. The PDG was initially used in various code optimizations, such as parallelism detection [54], code movement, or program slicing [113]. This representation can also support program comprehension and maintenance by depicting statement dependency trees in order to assess change impact [1], find code similarities [59, 61, 74, 126], measure code coverage, or reduce the test set for programs. Another graph-based code model that combines the Control Flow Graph (CFG) and Program Dependence Graph (PDG), along with an Abstract Syntax Tree, is known as the Code

Property Graph (CPG) [124]. This comprehensive model has been effectively employed in the detection of various vulnerabilities [11, 112].

Program Dependence Graph, Control Flow Graph and their derivative Code Property Graph, are limited to monolithic programs which means that analysis cannot cross the boundary of procedures. To overcome this drawback, Horwitz et al. [47] introduced the System Dependence Graph (SDG), augmenting the previous PDG model with edges representing dependencies between a call site and the called procedure, along with handling passed values via “procedure linkages.” Tracking values is not trivial to represent as a graph and requires defining several new abstract vertices [75]. The SDG allows for interprocedural code analysis, such as interprocedural slicing [47] or test case generation [56].

Over the years, object-oriented programming and other high-level programming languages have been popularized. Object-oriented programming introduced new concepts such as classes, methods, inheritance, and related abstract dependencies that are not represented in the basic System Dependence Graph. Researchers attempted to address this issue through SDG model extensions, as presented by Walkinshaw et al. [118] in the Java System Dependence Graph (JSysDG). JSysDG is a multigraph that represents both the structure of the program (method headers, classes, interfaces, and packages) and program behavior through SDG. Later, JSysDG was extended with object-flow dependence by Galindo et al. [41] to better facilitate program slicing. Shu et al. [106] proposed a practical implementation of the JSysDG model in JavaPDG—a platform for program dependence analysis along with a graphical viewer. However, the software solution was not open-sourced, and it is currently impossible to find and use it for pursuing new software dependency research. It is also worth mentioning that JSysDG builds on a similar concept to the Java Software Dependence Graph researched by Zhao [127].

Arora et al. [10] summarize various variations of Control Flow Graphs and Program Dependence Graphs. These models primarily aim to represent program execution flow or statement dependencies. The structure of the program, including abstract code elements like classes, is added on top of these graphs to facilitate more detailed program execution analysis and adapt to higher-level programming languages. This implies that software is analyzed at a very detailed level of statements, with a focus on understanding and optimizing program execution. However, for programmer, this level of detail may not be as critical with modern languages. Many sophisticated optimizations are handled directly by compilers or interpreters, reducing the programmer’s responsibility. Moreover, modern languages operate at higher abstraction levels, introducing abstract components and focusing on language expressiveness, better ways of structuring growing software, enabling component reuse, facilitating implementation replacements, and simplifying local refactoring activities. This shift in focus towards higher abstraction levels allows for faster program development, reduces the occurrence of certain errors, and enhances maintainability. Changing requirements, increasing software criticality, and the significant growth in codebase sizes have driven research efforts towards software structure comprehension and maintainability. The present aim is to reduce the time and cost associated with software maintenance rather than optimizing program execution.

To that end, software networks, also referred to as Artifact Dependency Graphs (ADG) [50], software collaboration graphs [79], or software architecture graphs [52] were researched. Software networks are general terms for abstract source code models that focus on code entities and their dependencies. They are not

intended to depict program execution but rather to describe the abstract graph model of the program's structure and dependencies. Myers [79] demonstrates that such graphs form scale-free, small-world networks and can be analyzed using a set of algorithms and metrics known from other complex network analyses [43, 52].

For object-oriented programming languages, software networks can consist of multiple levels of networks, such as file-level, package-level, class-level, or function-level collaboration networks [101]. The Class Collaboration Network (CCN) is particularly interesting to analyze [13, 31, 68, 119] because classes are the main building blocks in OOP. Arora and Goel [8, 9] proposed JavaRelationshipGraphs (JRG) — a software network model specialized for the Java language. JRG is a directed graph that captures various relations between different code elements in Java programs. In this graph, nodes represent packages, classes, interfaces, methods, attributes, and edges represent class extensions, implementations and aggregations, imports, ownership, and method calls. There is not one definitive guide on how to create such a software network for a particular language, but there is an attempt made by Savic et al. [101] to create a language-independent model of dependencies between source code entities and establish a General Dependency Network (GDN). While this is a promising direction, it may miss some language-specific details and GDN is extracted from yet another custom model — the extended abstract syntax tree (eCST), which complicates model adoption and potential implementations.

The Call Graph [98] is another popular network wherein the graph of caller-callee relationships is extracted from the source code. The main goal of the Call Graph is to support static and dynamic analysis of the code call dependency flow without concerning the detailed code structure. Grove et al. [44] distinguish two types of the Call Graph – the context-insensitive one, in which one node uniquely represents one procedure, and the context-sensitive one, in which each procedure call is represented as a separate node. For code navigation and depicting the graph structure, the context-insensitive approach is more natural, as each definition in the code has precisely one unique node representation, similar to the source code definition in the file. Various papers use call graphs for code analysis [3, 46, 60, 64]. Although proven useful, the Call Graph is not a complete representation of the code dependencies as it contains only information about chains of calls.

Software networks are extensively used to facilitate software comprehension and maintenance through software architecture recovery [70, 102], finding key entities [31, 68, 76, 86], code modularization [50, 91, 110], finding code similarities [114], code visualization [19, 71, 90], and tracking code evolution [15].

The proposed Semantic Code Graph (SCG) is a form of software network. We conduct a detailed comparison of the SCG with other models in Section 5. In particular, we highlight the limitations of existing models concerning software comprehension. **To our knowledge, no study has yet compared graph code representations in this context.**

2.2 Tools for extracting code dependency structure

To leverage the progress made in software comprehension research related to program graph representation, the utilization of readily available code dependency graph representation tools is essential. Furthermore, to facilitate ongoing research and establish persuasive comparisons, the importance of publicly accessible graph extraction solutions is crucial. These solutions should export graph models in an easily digestible in-

intermediate data representation to allow further creation of tools directly supporting software comprehension activities.

For the Java language, there is body of analysis tools which operate on the JVM bytecode and allow also for extracting some of the code dependency representations. *Soot* [63, 116], and its successor *SootUp*¹, although primarily created for introducing bytecode optimizations, can generate Control Flow Graphs (for particular methods or classes) and extract Call Graph for the project. *SootUp* operates on the JVM bytecode through the Jimple intermediate representation which aims to represent JVM bytecode in a human readable way. Jimple mimics the JVMs class file structure and allows user to inspect and conduct various Data Flow, Data Dependency or Call Graph analysis through the Java API.

Another well-known software is *WALA*² which enables interprocedural analysis of Class Hierarchy and Call Graph. It also defines WALA Intermediate Representation encoding instructions and control flow of a particular method.

*JDepend*³ can generate predefined design quality metrics for each Java package in terms of extensibility, reusability, and maintainability. However it does not focus on software comprehension activities and as such, it does not expose any way to export data that could be used later for further analysis. Moreover, the last release of the software was in 2020⁴, and since then it does not seem to be actively maintained.

*Dependency Finder*⁵ can find and export Java simplified dependencies from the bytecode representation, but it can only extract three types of dependencies: class-to-class, feature-to-class, and class-to-feature (where feature represents everything but a class). *Dependency Finder* can export dependencies to a TXT or XML format. However, aside from the fully qualified name of the entity, no other details are stored in this model.

*jdeps*⁶ is another tool capable of presenting Java dependencies extracted from bytecode. However, these dependencies are limited to only package and class levels and do not consider other code entities or precise relations between them.

The tools such as *SootUp*, *WALA*, *Dependency Finder* or *jdeps* rely on the JVM bytecode analysis and have all the drawbacks of the bytecode representation, such as missing type information due to the type erasure, missing precise source code location or presence of additional artifacts generated by the compiler to implement advanced language features using instructions of the JVM bytecode. We describe the JVM bytecode limitations in detail in Section 3.4.5.

There are also tools that operates directly on the source code of Java program, such as *Spoon* [87] or *JavaParser* [109]. *Spoon*⁷ can analyze, rewrite and transform Java source code. Similarly *JavaParser*⁸ can analyze, transform and generate Java source code. However, both libraries does not expose any particular dependency graph source code representation out of the box. Such capabilities has to be implemented man-

¹<https://soot-oss.github.io/SootUp/>

²<https://github.com/wala/WALA>

³<https://github.com/clarkware/jdepend>

⁴<https://github.com/clarkware/jdepend/releases>

⁵<https://depfind.sourceforge.io/>

⁶<https://docs.oracle.com/en/java/javase/11/tools/jdeps.html>

⁷<https://github.com/INRIA/spoon>

⁸<https://github.com/javaparser/javaparser>

ually. In fact, we facilitate *JavaParser* advanced functionalities to implement SCG extractor program for Java projects.

2.3 Software comprehension tools

Some existing tools provide software comprehension capabilities. *Scitools Understand*⁹ allows for various static code analyses, code dependency visualization, and advanced code browsing. However, *Scitools Understand* is a paid tool for commercial usage and can only export basic class-level and file-level software networks.

Code Compass [90] is an open-source primarily LLVM/Clang-based tool developed to augment the understanding of large legacy C/C++ systems. Currently, it supports also Java language by using the Eclipse JDT parser¹⁰. It supports Call Graph and Class Collaboration Networks visualization and offers browsing capabilities through the web UI, but not exporting data for further analysis and data mining in external tools.

Doxygen is a tool primarily created to generate documentation from annotated C++ sources, but it can also support Java documentation (and other popular languages). Apart from generating documentation from documented sources it can also extract the code structure for entities which are undocumented, as well as visualize the relations between various elements through dependence graphs, inheritance diagrams or collaboration diagrams. Such diagrams are generated per class level and exported in the .DOT format with limited detail. However, there is no straightforward method to extract the complete dependency graph for the entire project, or perform more advanced software comprehension activities directly through the *Doxygen* tool.

Wheeldon and Counsell [119] in their work extract a detailed class collaboration network with the *AutoCode* software, but its actual implementation is not available.

*Moose*¹¹ is a platform based on *Pharo*¹² that enables custom analysis of software systems. The software model has to be extracted in MSE format [33] using external tools, such as VerveineJ¹³. The MSE format accurately represents the exact source code. Although *Moose* provides a powerful environment for detailed analysis facilitated through Smalltalk-like queries, it has a steep learning curve and does not offer built-in support for popular code graph models. Consequently, significant implementation effort is needed to support common software comprehension activities, such as software architecture discovery, modularization, or the identification of critical project entities in a manner that is quick and approachable for inexperienced users.

In summary, based on our current knowledge, **there is no publicly available and free to use tool to extract and comprehend a rich System Dependence Graph or Software Network Graph for the Java or Scala languages.**

⁹<https://scitools.com/>

¹⁰<https://eclipse.dev/jdt/>

¹¹<http://moosetechnology.org/>

¹²<https://pharo.org/>

¹³<https://github.com/moosetechnology/VerveineJ>

Chapter 3

Semantic Code Graph

3.1 Source code dependency model requirements

The goal of the Semantic Code Graph information model is to preserve the direct relation to the source code (syntax) while also capturing the meaning (semantics) of the source code elements and dependencies between them. An information model that meets such requirements will facilitate source code analysis in various areas. Here we particularly focus on three of them:

- *Code structure comprehension*, e.g., learning code structure by interactive code visualization and browsing, applying data analysis to find the most important code elements, answering reachability questions [64], recovering software architecture [42] or facilitating semantic code search [48].
- *Software quality assessment*, e.g., measuring software metrics through graph properties [37,52,92,96].
- *Software refactoring*, e.g., supporting software modularization through suggested code partitioning, suggesting method movements, or code structure improvements [25,32,84].

The important ‘direct relation to source code’ requirement simply denotes that the entities seen by the developer while working with the source code are represented in the model as closely as possible. Consequently, the SCG model should be extracted either directly from the source code, or another representation that closely matches the source code, e.g. the Abstract Syntax Tree (AST). Extraction from an intermediate representation, such as bytecode – while tempting for various reasons – would not be appropriate since such a representation contains entities generated by the compiler and lacks important details because of early optimizations (see Section 3.4.5 for more in-depth analysis of this). Moreover, all entities and relations between entities should have the *location* property associated with them. This is a crucial property for building interactive visualization, or for pointing the developer to the exact place in the source code when performing static analysis or preparing any potential code manipulations. Any advice presented by a tool – such as presenting visual structure, found problems, extracted knowledge, or improvement suggestions – should always precisely point the user to an appropriate place in the source code.

In addition, each extracted entity and relation should retain as much semantic information as possible. For example, an entity representing a method should also contain its scope (private, public), modifiers (abstract, override, etc.), the number of arguments and their types, return types, etc. This will allow for precise,

close-to-source-code model analysis focused on dependencies between source code entities, at the same time leveraging most of details available in the AST code representation.

3.2 Semantic Code Graph base model

Semantic Code Graph (SCG) is a source code information model aimed at representing diverse dependencies present in the source code, with the objective of fulfilling the requirements outlined in Section 3.1. The SCG data structure is a graph where nodes denote code declarations and definitions, while the edges represent various dependencies between them. The SCG is formally defined as a pair $G_{SCG} = (V_D, E_R)$ comprising:

- $V_D = \{v_1, v_2, \dots, v_n\}$, a set of vertices (nodes), where v_i represents a code entity of type *CLASS*, *OBJECT*, *TRAIT*, *INTERFACE*, *METHOD*, *PARAM*, *TYPE_PARAM*, *VALUE*, *VARIABLE*, *TYPE*, *ENUM*. This list of node types is flexible and can be extended with other entities specific for a given language; n is the total number of entities;
- $E_R = \{(v_i, v_j) \mid v_i, v_j \in V_D\}$, a set of directed edges comprising ordered pairs of nodes, and representing *relations* between these nodes. The possible types of edges include *CALL*, *DECLARATION*, *EXTEND*, *OVERRIDE*, *PARAMETER*, *RETURN_TYPE*, *TYPE_PARAMETER*. Similar to nodes, new edge types can be added if required to capture specific code dependency relations occurring in a language.

The basic data structures of the SCG graph, as a foundation for diverse semantic information about source code dependencies, are rather simple. In particular, each node has to contain the following:

- *id* – unique node identifier,
- *kind* – type of the node (one of the node types mentioned in the V_D definition above),
- *location* – place in the source code where the entity represented by this node is defined,
- *displayName* – node name meaningful to the end user,
- *edges* – set of outgoing edges.

The edges in the SCG graph are described by the following properties:

- *to* – node identifier to which this edge is pointing to,
- *location* – place in the source code where the relation represented by this edge can be observed,
- *type* – the type of the edge (its label); *type* is one of the edge types mentioned in the E_R definition above.

The *location* property has its own structure that provides precise information about the position of a node or edge in the source code file. It comprises the following fields:

- *uri* – the relative path to the source code file,

- *startLine* – indicates the starting line number for the position of the node or edge in the file,
- *startCharacter* – specifies the starting character position for that node or edge,
- *endLine* – indicates the end line number for the position of that node or edge in the file,
- *endCharacter* – specifies the ending character position for that node or edge.

The SCG nodes represent all concrete entities that describe the essential code structure. In particular, nodes do not represent abstract or arbitrary code organization entities, such as packages, files, or modules. The information about these aspects (e.g. that a class belongs to a package or is defined in a certain file) is very useful, but it is better to provide it as additional node or edge properties, rather than essential code structural elements, as it would disrupt the representation of the software structure and quality analysis of the code.

In addition to essential properties, each node and edge can contain various additional properties encapsulated in the *properties* map field. These properties may encompass information such as the number of lines of code (LOC), the package to which the entity belongs, its visibility scope, various modifiers, or any language-specific details that prove useful for subsequent analysis. Furthermore, properties can be extracted from sources other than the source code itself, for example, from the source control version system, to acquire details such as the last edit timestamp, the primary author, the latest commit message, and so forth. However, it should be noted that mapping certain source control version information into SCG nodes is not straightforward because most source control versions operate at the level of entire lines. When dealing with classes, extracting the author of the most recent changes becomes feasible by utilizing the class declaration location and the property indicating the number of lines of code within the class. This allows us to search for the last change within the class's defined scope. For one-line definitions such as variables or parameters, we can take advantage of the token-based git history approach introduced by the Context Buddy tool¹. This tool explores the git history from the project's inception and, among other things, extracts and tracks the introduced tokens, token history, and the original authors of the tokens. Token history can be easily matched with a specific node, thanks to the node's *location* property.

The specific node kinds and edge types, along with their connections, are language-specific details that should be implemented in a manner natural and most suitable for each programming language. In the following sections, we describe specific implementations of the SCG model for the Scala and Java languages.

3.3 Semantic Code Graph for Scala and Java

Scala [82] is a hybrid language that combines constructs known from object-oriented and functional programming. Scala 2 and its next iteration Scala 3, themselves based on JVM, were designed to supersede the Java language in its capabilities, extending and improving on already proven useful features rather than completely changing the paradigm. As a result, Java and Scala share similar entity types and relations in terms of the Semantic Code Graph, as presented in Table 3.1.

¹<https://github.com/VirtusLab/contextbuddy/tree/master/intellij>

Java/ Scala	Node kind	Description	Example
J,S	<i>CLASS</i>	a class	class A
S	<i>OBJECT</i>	an object	object A
S	<i>TRAIT</i>	a trait	trait B
J	<i>INTERFACE</i>	an interface	interface B
J,S	<i>METHOD</i>	a method	def m(): A
J,S	<i>PARAM</i>	method, constructor or trait parameter	a in def m(a: A)
J,S	<i>TYPE_PARAM</i>	a type parameter	T in def m[T](t: T)
J,S	<i>VALUE</i>	value in Scala or final variable in Java	val a: A
J,S	<i>VARIABLE</i>	a variable (non-final field)	var a: A
S	<i>TYPE</i>	type member	type T <: A
J,S	<i>ENUM</i>	enumerated type	enum E { case A }
Java/ Scala	Edge type	Description	Example
J,S	CALL	call to method, constructor; usage of parameter or field	$\text{def } m1() = m2() \Rightarrow m1 \xrightarrow{\text{CALL}} m2$
J,S	DECLARATION	relation between parent and declared entity	$\text{class } A \{ \text{def } m() = () \} \Rightarrow A \xrightarrow{\text{DECLARATION}} m$
J,S	EXTEND	extend relation between entities	$\text{class } A \text{ extends } B \Rightarrow A \xrightarrow{\text{EXTEND}} B$
J,S	OVERRIDE	edge from (implementation) method to its overridden method	$\text{override def } m() \Rightarrow A \# m \xrightarrow{\text{OVERRIDE}} B \# m$
J,S	PARAMETER	edge from entity to its parameter	$\text{def } m(a: A) \Rightarrow m \xrightarrow{\text{PARAMETER}} a$
J,S	RETURN_TYPE	edge from method to its return type	$\text{def } m(): B \Rightarrow m \xrightarrow{\text{RETURN_TYPE}} B$
J,S	TYPE	type relation e.g. between variable definition and its type	$\text{val } a: A \Rightarrow a \xrightarrow{\text{TYPE}} A$
J,S	TYPE_PARAMETER	type parameter	$T \text{ in } \text{def } m[T](t: T) \Rightarrow m \xrightarrow{\text{TYPE_PARAMETER}} T$

Table 3.1: List of nodes and edges present in SCG model for Java and Scala

Fig. 3.1 and Fig. 3.2 show source code samples and their respective SCG representations. Fig. 3.1 presents two top-level declarations of class *B* and trait *A*. The respective SCG graph captures various dependencies in this piece of code depicting each declared entity – class *B* with method *bar()*, trait *B* with method *foo()*, and existing dependencies between them of type *DECLARATION*, *EXTEND* and *CALL*.

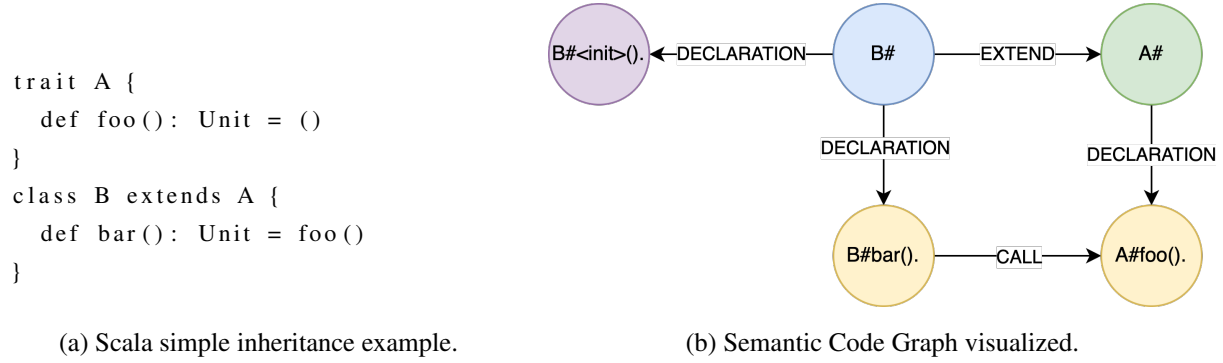


Figure 3.1: SCG visualization for a simple inheritance example.

Fig. 3.2 shows method *triple* with one parameter *n* and local value *t* which is computed as $n * n * n$. Consequently, the *n* parameter of method *triple* is called by this method 3 times. The *t* parameter is then called and returned in line 3. Type *Int* is deliberately not represented in the SCG as its definition exists only in an external library. SCG represents only entities defined within the code base of the analyzed project.

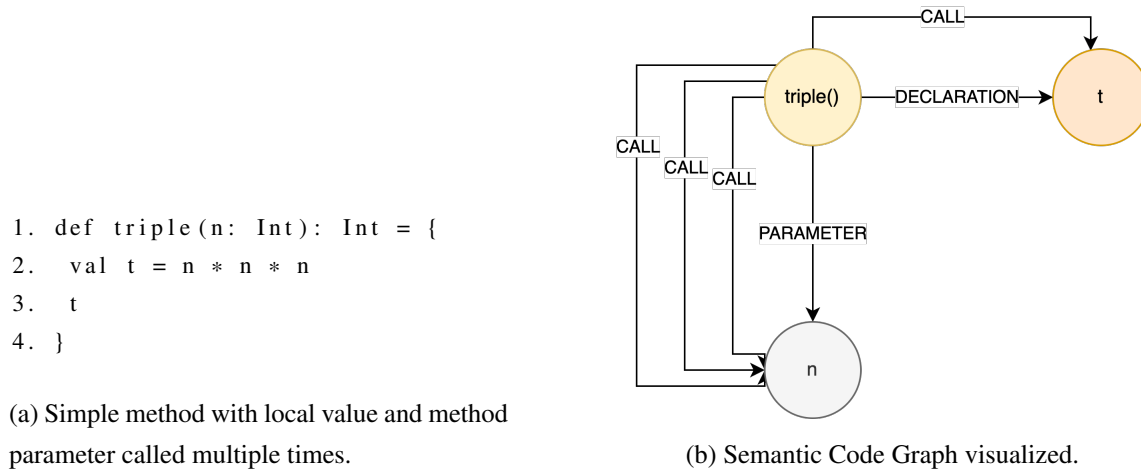


Figure 3.2: SCG visualization of method with local calls and local value *t* definition.

3.4 Semantic Code Graph extraction process

3.4.1 Implementation overview

A common way of extracting the structure of the source code is by parsing it into an Abstract Syntax Tree (AST). However, the AST represents only syntax, lacking the semantic information about relations between code entities, required by the SCG (Section 3.2). Also, the AST describes entities not important from the semantic code dependency point of view, such as braces, keywords, and comments. To build a comprehen-

sive SCG model from an AST, we therefore need to drop unnecessary information from the AST graph and augment the remaining nodes with semantic information. Adding relations between entities will create a graph of semantic and structural dependencies.

Semantic analysis is one of the crucial steps in the compilation process [45]. Consequently, useful semantic information required by the Semantic Code Graph can be extracted from the information generated during the semantic analysis performed by the compiler in the front-end analysis phase of the compilation. All steps of the front-end analysis are schematically depicted in Fig. 3.3. In the *Hierarchical Analysis* step, the parser consumes code tokens and produces an Abstract Syntax Tree, a lossless representation of the source code, preserving all its syntactical details. In the next step, the semantic analyzer augments the AST with additional semantic and contextual information with the help of *Symbol Tables*. Symbol Tables facilitate efficient insertion and lookup of code elements by name, reflecting code dependencies, e.g., linking references to their corresponding declarations. *Semantic Analysis* (also known as context-sensitive analysis) annotates the AST with types, references, scopes, and other semantic attributes. Such an AST, with all types and symbols resolved, is a sufficiently detailed abstraction for generating SCG.

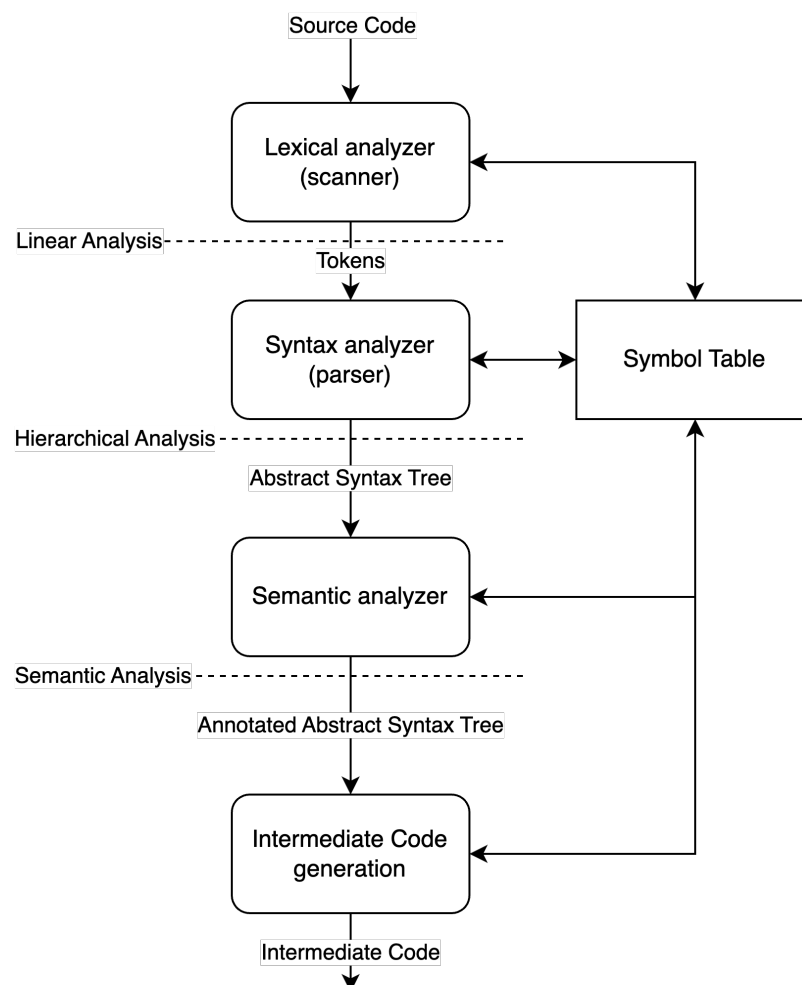


Figure 3.3: Front-end analysis steps in a compiler

3.4.2 Extracting SCG for the Scala2 language

The Scala 2 compilation process consists of 24 phases (Fig. 3.4), of which only the first four (*parse*, *namer*, *packageobjects* and *typer*) are required to extract information useful for the SCG. To this end, we can create a compiler plugin and work directly with the compiler internal structures. However, compilation is a complex process, so dealing with compiler internal structures can be challenging. An alternative solution is to use other meta-tools implemented to facilitate Scala 2 syntactic and semantic analysis. An untyped AST can be generated with the *scalameta Trees*² library and then annotated with semantic information available through the *SemanticDB*³ library, which exposes symbols and types defined in the source code. *SemanticDB* is a compiler plugin that is attached after the *typer* phase. We can use these two libraries and create our own Scala plugin⁴ capable of extracting SCG, also invoked after the *typer* compilation phase.

```
$ scalac -Xshow-phases
  phase name  id  description
  -----
      parser   1  parse source into ASTs,
                  perform simple desugaring
      namer    2  resolve names,
                  attach symbols to named trees
packageobjects 3  load package objects
      typer    4  the meat and potatoes:
                  type the trees
      ...
      jvm     23  generate JVM bytecode
terminal      24  the last phase
                  during a compilation run
```

Figure 3.4: Scala 2 compiler phases. All type related information are available after the *typer* phase.

3.4.3 Extracting SCG for the Scala3 language

The Scala 3 compiler introduces the Typed Abstract Syntax Tree (TASTy) [83], an annotated AST serving as an intermediate code between the front and back end of the compiler. TASTy contains a complete source code syntax and resolved semantic details (such as types, references, and others). The TASTy trees are stored as binary files **.tasty* during the *pickler* compilation phase (Fig. 3.5). Instead of writing our own compiler plugin, we can read these files from the compiled project and generate the SCG in an external process.

3.4.4 Extracting SCG for the Java language

The extension mechanism for the *javac* compiler was introduced in Java 8. Over the years, writing compiler plugins and dealing with compiler internals has proven to be challenging and subject to internal API changes.

²<https://scalameta.org/docs/trees/guide.html>

³<https://scalameta.org/docs/semanticdb/guide.html>

⁴<https://github.com/VirtusLab/scg-scala>

phase name	description
-----	-----
parser	scan and parse sources
typer	type the trees
...	
pickler	generates TASTy info
...	
genBCode	generate JVM bytecode

Figure 3.5: Scala 3 compiler phases, with *pickler* phase for storing TASTy binary files.

To facilitate code analysis that does not require full features of the compiler, the *JavaParser*⁵ library was created. *JavaParser* not only produces the AST, but also (since version 3.5.0) resolves semantic dependencies through a special module called *SymbolSolver*. This feature can be utilized to extract the SCG from the Java source code. The library supports all language constructs from Java 1 to Java 17 and is actively developed, which gives us great coverage of language features with minimal effort. We have enhanced our SCG command line tool, *scg-cli* (introduced in Chapter 7), with the functionality to extract SCG data for Java projects. This feature can be accessed using the command `scg-cli generate <path/to/project>`.

3.4.5 Challenges in extracting SCG from Bytecode

Sections 3.4.2-3.4.4 described the implementation of the SCG extraction for three different languages: Scala 2, Scala 3 and Java. All three of them, with the help of compilers during the back-end analysis, are translated to the bytecode – the Intermediate Code representation for the Java Virtual Machine. Consequently, it would be beneficial from the implementation point of view to extract SCG from the JVM bytecode as this would automatically support any JVM-based language. In Section 3.4.1, we discussed how the compiler front-end provides the most convenient and accurate representation of the source code for our objectives. On the contrary, bytecode is an output from the compiler back-end process which involves various optimizations and simplifications applied to generate an output optimized for the JVM interpreter. Indeed, the accuracy of the model extracted from bytecode suffers from representation mismatch. Let us consider a few notable examples:

- Missing information on precise element location in the source code.
- Missing type information due to the type erasure mechanism.
- Additional JVM bytecode generated to cover specific language features not represented in bytecode.
- Various compiler optimizations.

⁵<http://javaparser.org/>

Missing location data

Java programs are compiled with the *javac* compiler. The resulting bytecode contains only limited information on the location of the software entities. Even with the *-g* parameter, the generated bytecode does not contain location for non-executable code such as abstract methods (Fig. 3.6).

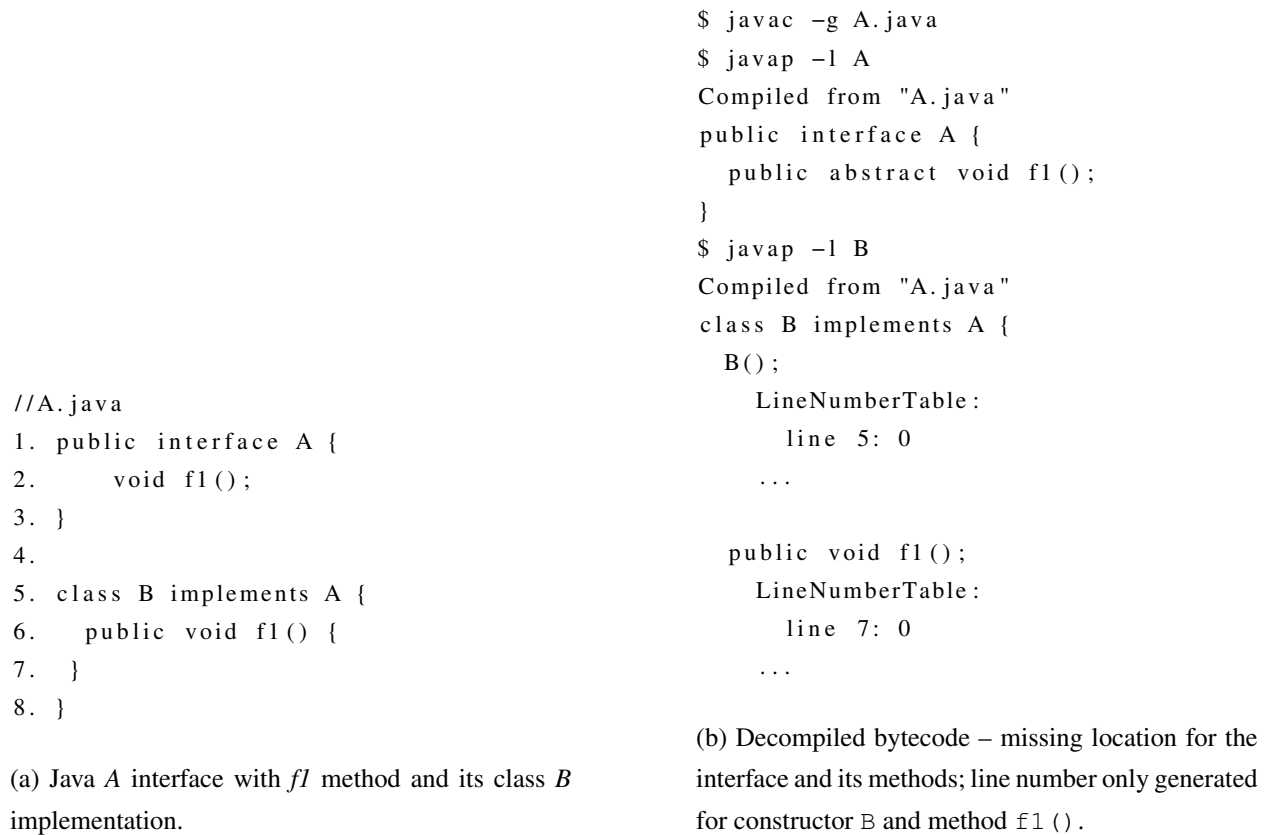


Figure 3.6: Java bytecode does not contain the source code location of non-runtime code elements even with the *javac -g* flag. For other elements only the line number can be recorded.

Missing type data

In Java, we can define generic types, such as *TypeErasure*<A>, but the actual type will be lost in the compilation due to type erasure mechanism⁶ (Fig.3.7).

Extra compiler-generated bytecode

The problem of inadequate JVM bytecode is especially visible for languages other than Java. For example, Scala has a *trait*⁷ code structure that is similar to a Java interface, but it can have concrete fields and methods, and provides the multiple inheritance capability with the help of the Scala trait linearization algorithm (Fig. 3.8).

⁶<https://docs.oracle.com/javase/tutorial/java/generics/erasure.html>

⁷<https://docs.scala-lang.org/tour/traits.html>

<pre>//TypeErasure.java class TypeErasure<A> { public A a; public TypeErasure(A a){ this.a = a; } } //M.java public class M { void main(String[] args) { TypeErasure<Integer> a = new TypeErasure<Integer>(1); System.out.println(a.a + 1); } }</pre>	<pre>//TypeErasure.class class TypeErasure<A> { public A a; public TypeErasure(A var1) { this.a = var1; } } //M.class public class M { public M() { } public void main(String[] var0) { TypeErasure var1 = new TypeErasure(1); System.out.println((Integer)var1.a + 1); } }</pre>
(a) <i>TypeErasure</i> <A> generic type	(b) Decompiled bytecode

Figure 3.7: Java source code: The *Integer* type for the generic *TypeErasure* class will not be represented in bytecode for the local variable *a*.

<pre>trait A { def foo(): Unit = () } class B extends A { def bar(): Unit = foo() }</pre>	<pre>// A.class public interface A { static void foo\$(final A a) {a.foo();} default void foo() {} static void \$init\$(final A a) {} } // B.class public class B implements A { public void foo() { A.foo\$(this); } public void bar() { this.foo(); } public B() {A.\$init\$(this);} }</pre>
(a) Scala source code: class B extends trait A.	(b) Decompiled compiler-generated bytecode.

Figure 3.8: The Scala compiler generates the *B#foo()* method with reference to the generated static method *A#foo\$(a)*.

Even in very simple cases, the call graph generated from the Scala bytecode can be very different from one that actually occurs in the source code, as presented in Fig. 3.9.

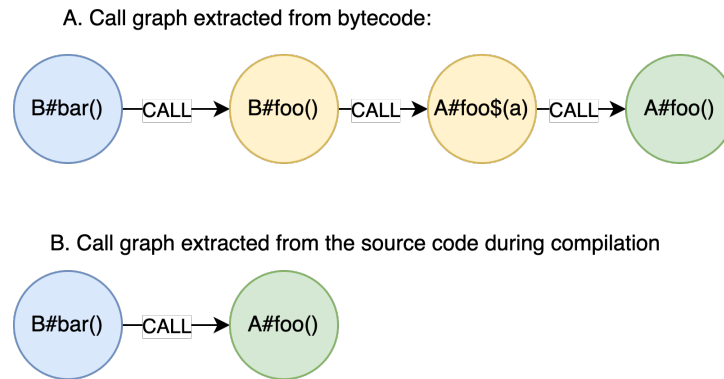


Figure 3.9: Comparison of a call graph generated from bytecode and from the source code – *B#foo()* and *A#foo\$(a)* methods do not exist in the source code.

Such extensive changes to the generated bytecode result in a mismatch between the source code and its representation in the bytecode [5, 6]. Consequently, accurate information about code dependencies can only be extracted directly from the source code rather than using an intermediate code representation such as bytecode.

3.4.6 Stable symbol identifiers for a multi-language SCG model

To create bindings between references and definitions, and reflect them in a graph structure, we need to adopt conventions for constructing global stable identifiers for code entities. The same identifier has to be generated when analyzing a definition and when analyzing a reference to this definition. The unique symbol identifier can also be used as a node identifier and to construct appropriate relations between nodes in the SCG. Symbol names are unique only within a given context, and some names might be shadowed in an inner context. To create unique identifiers, we need to combine the symbol name with that symbol’s owner identifier. In this way, we can ensure the uniqueness of each symbol identifier. We propose a set of rules⁸ presented in Table 3.2, using which we can construct a stable identifier for any language symbol.

3.4.7 SCG storage format

It is essential to store the SCG data in a format optimized for performance, both during data generation and analysis. We have identified key attributes that such a format should support:

1. *Incremental builds* – some software projects may contain millions lines of code, therefore the format has to allow building the graph incrementally during the generation process [29].

⁸partially adopted from <https://scalameta.org/docs/semanticdb/specification.html#symbolinformation-1>. Access 22.05.2023.

Lang	Symbol type	Rule	ID
J,S	<i>PACKAGE</i>	name + /	<i>p/</i>
J,S	<i>CLASS</i>	Owner ID + #	<i>p/A#</i>
S	<i>OBJECT</i>	O. ID + .	<i>p/B.</i>
S	<i>TRAIT</i>	O. ID + #	<i>p/T#</i>
J,S	<i>METHOD</i>	O. ID + name + (.	<i>p/B.mB(.</i>
J,S	<i>METHOD*</i>	O. ID + name + (+n).	<i>p/B.mB(+1).</i>
J,S	<i>PARAM</i>	O. ID + (name)	<i>p/A#mA(.(a)</i>
J,S	<i>TYPE_PARAM</i>	O. ID+ [name]	<i>p/A#mT().[T2]</i>
J,S	<i>VALUE</i>	O. ID + name + .	<i>p/B.b.</i>
J,S	<i>VARIABLE</i>	O. ID + name + (.	<i>p/B.c(.</i>
S	<i>TYPE</i>	O. ID + name + #	<i>p/B.T#</i>

Table 3.2: Rules for creating stable identifiers in the Java and Scala languages (related code containing all symbol types is presented in Listing 3.1). *Note that overloaded methods are tagged with a position number.

Listing 3.1: Scala code reflecting stable identifiers presented in Table 3.2

```
// A.java
package p
class A {
  def mA(a: String): String = a
  def mT[T2](t: T2): Unit = {}
}

// B.scala
package p
object B {
  type T = String
  val b = "b"
  var c = "c"
  def mB(a: A): String = {
    a.mA(b)
  }
  def mB(a: String): String = {
    ""
  }
}
```

2. *Partial graph loading* – exported graphs may contain hundreds of thousands of vertices, so it would be inefficient to load the entire graph in the case the user needs to perform analysis only on a particular module.
3. *Minimum disc space usage* – disk usage is important as it will be desirable to distribute graph files in the control version system or as an attachment to the particular code base.

4. *Extensibility* – To support various programming languages, we need to be able to add different properties to the graph, so the format needs to be easily expandable.
5. *Libraries in multiple programming languages* – Code generating the SCG data for a given language will typically be written in the same language. On the other hand, the code to analyze the SCG data may be written in different suitable languages, such as Python or R. Consequently, it is crucial to provide APIs to write and read the format in multiple programming languages.
6. *Conversion to other data formats* – for more advanced data analysis we need to be able to convert the SCG representation to other data formats, such as data frames, or graph file formats specific for existing tools, e.g., as GDF⁹ or DOT¹⁰.

Many different graph formats were proposed for exporting huge graph structures [125], but none of them satisfies all our requirements. Consequently, we propose a new graph format based on the Protocol Buffers¹¹. For the Semantic Code Graph this format can be treated as an intermediate data exchange format between the source code and a data analysis tool, or a target graph format file. To address the above requirements, we made the following design choices:

- store one graph file per source file in the project (requirements 1 and 2),
- adopt protobuf as the base format having a small footprint (requirement 3), and being supported by many languages (requirement 5),
- schema definition contains a flexible property map with any number of arguments (requirement 4).

Requirement 6 (conversion to other data formats) can easily be fulfilled by writing dedicated converters which is facilitated by utilizing Protocol Buffers, supported by all major languages¹².

The complete schema for the proposed format is presented in Listing 3.2. The format defines data structures for graph nodes, edges and location in the code, grouped together under one structure representing the entire source code file. Properties of the SCG, as defined in Section 3.2, are mapped to individual fields of these structures. Additional dynamic node and edge properties can be placed in the dictionary structure *properties* – it is a single format extension point for any additional, often language-specific, properties.

The files are saved with the `.semanticgraph` extension.

Listing 3.2: Protobuf Semantic Code Graph schema definition.

```
syntax = "proto3";

message Location {
    string uri = 1;
    int32 startLine = 2;
    int32 startCharacter = 3;
```

⁹<https://www.iso.org/standard/54610.html>

¹⁰<https://graphviz.org/doc/info/lang.html>

¹¹<https://developers.google.com/protocol-buffers>

¹²<https://protobuf.dev/reference/other/>

```
    int32 endLine = 4;
    int32 endCharacter = 5;
}

message Edge {
    string to = 1;
    string type = 2;
    Location location = 3;
    map<string , string> properties = 4;
}

message GraphNode {
    string id = 1;
    string kind = 2;
    Location location = 3;
    map<string , string> properties = 4;
    string displayName = 5;
    repeated Edge edges = 6;
}

message SemanticGraphFile {
    string uri = 1;
    repeated GraphNode nodes = 2;
}
```

Chapter 4

Outline of research objectives and methods

This Chapter provides a concise outline of our research objectives and methods, and serves as a roadmap for subsequent chapters evaluating the Semantic Code Graph model.

4.1 Research objective

The primary research objective of this dissertation is to assess the effectiveness of the Semantic Code Graph (SCG) model in enhancing software comprehension capabilities. To achieve this goal, we compared the SCG model with two other source code representation models: the Class Collaboration Network (CCN) and the Call Graph (CG). The research questions guiding this study are as follows:

- **RQ1:** Does the SCG model enhance software comprehension capabilities in comparison with the Class Collaboration Network and the Call Graph models?
- **RQ2:** Does SCG-based data analysis enable actionable software comprehension insights?

4.2 Selection of models

In Chapter 5, we evaluate the SCG against nine other source code graph representation models. The selected models are as follows: Control Flow Graph (CFG) [7], Program Dependence Graph (PDG) [38], Code Property Graph (CPG) [124], System Dependence Graph (SDG) [47], Java System Dependency Graph (JSysDG) [118], Call Graph (CG) [98], Class Collaboration Network (CCN) [119], Java Relationship Graphs (JRG) [8], and General Dependency Network (GDN) [101].

A comprehensive comparison conducted from the software comprehension perspective and employing 14 distinct criteria, has led us to choose two models for further investigation: the Call Graph (CG) and the Class Collaboration Network (CCN). The rationale behind this decision is that these two models are notably well-established among researchers and they offer two unique perspectives on software: structural (CCN) and operational (CG). Thus, utilizing these models for software comprehension may provide diverse insights into the software. Given these factors, we have chosen these models for a thorough empirical comparison with our proposed SCG model.

4.3 Data collection

For empirical evaluation, we extract SCG, CCN and CG models from eleven open source projects. The selected projects are diverse in terms of the size, complexity, and application domains, providing a robust dataset for the evaluation of the SCG model. Moreover, the projects are very popular among programmers – nine out of eleven have more than 10k stars on the GitHub repository. To enable results reproduction, we have also published the extracted SCG models¹.

The eleven chosen projects are as follows:

- retrofit² – a type-safe HTTP client for Android and Java,
- commons-io³ – the Apache Commons IO library containing utility classes, stream implementations, file filters, file comparators, endian transformation classes,
- playframework⁴ – high velocity web framework for building web applications with Java & Scala,
- metals⁵ – Scala language server with rich IDE features,
- glide⁶ – media management and image loading framework for Android that wraps media decoding, memory and disk caching, and resource pooling into a simple and easy to use interface,
- vert.x⁷ – toolkit to build reactive applications on JVM; Vert.x Core library containing low-level functionalities including support for HTTP, TCP and file system access,
- RxJava⁸ – Reactive Extensions for the JVM; a library for composing asynchronous and event-based programs by using observable sequences,
- dubbo⁹ – Apache Dubbo, a high-performance, Java-based open-source RPC framework,
- spring-boot¹⁰ – popular library to help with creating Spring applications,
- Akka¹¹ – a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala,
- Spark¹² – a well-known engine for large-scale data analytics.

¹The collected data is available at <https://github.com/VirtusLab/scg-cli/tree/main/data>

²<https://github.com/square/retrofit>

³<https://github.com/apache/commons-io>

⁴<https://github.com/playframework/playframework>

⁵<https://github.com/scalameta/metals>

⁶<https://github.com/bumptech/glide>

⁷<https://github.com/eclipse-vertx/vert.x>

⁸<https://github.com/ReactiveX/RxJava>

⁹<https://github.com/apache/dubbo>

¹⁰<https://github.com/spring-projects/spring-boot>

¹¹<https://github.com/akka/akka>

¹²<https://github.com/apache/spark>

4.4 Tools

To demonstrate the practical usability of the SCG and to support further research, we have developed and released two ready-to-use products: the *Graph Buddy* IDE plugin and the *scg-cli* command-line tool. Both of these tools utilize SCG as an underlying data information model.

Graph Buddy is a code dependency browsing and visualization tool available on the official marketplace for IntelliJ¹³ and VSCode¹⁴ IDEs. We provide a detailed description and evaluation of *Graph Buddy* in Chapter 6.

The *scg-cli* is a command-line tool designed to support various software comprehension activities within specific projects. The tool's source code and executable files are available on the GitHub repository¹⁵. We describe the software and demonstrate its capabilities in Chapter 7. Furthermore, we demonstrate the practical value of *scg-cli* by utilizing the tool's capabilities in our empirical evaluation.

4.5 Empirical evaluation

In Chapter 8, we conduct an empirical comparison of the SCG model with two other source code representation models: the Class Collaboration Network (CCN) and the Call Graph (CG). We address the research questions through a comparative and qualitative analysis of these selected models across various software comprehension activities performed on eleven open-source projects described in Section 4.3. The activities include:

- comprehension of project structure,
- recovery of software architecture,
- identification of critical project entities,
- discovery of code similarities.

¹³<https://plugins.jetbrains.com/plugin/13467-graph-buddy>

¹⁴<https://marketplace.visualstudio.com/items?itemName=virtuslab.graph-buddy>

¹⁵<https://github.com/VirtusLab/scg-cli>

Chapter 5

Software Comprehension capabilities: comparison of SCG and other graph-based code representations

5.1 Introduction

In this section, we conduct a comparative analysis of various models as outlined in Chapter 2, contrasting them with the SCG model within the context of software comprehension. We consider nine different code graph models in addition to the Semantic Code Graph (SCG): Control Flow Graph (CFG) [7], Program Dependence Graph (PDG) [38], Code Property Graph (CPG) [124], System Dependence Graph (SDG) [47], Java System Dependency Graph (JSysDG) [118], Call Graph (CG) [98], Class Collaboration Network (CCN) [119], Java Relationship Graphs (JRG) [8], and General Dependency Network (GDN) [101].

Our evaluation focuses on two key aspects: the models' suitability in representing crucial features from the perspective of code dependency software comprehension and their readiness for practical adoption in commercial settings. We break down the analysis into three categories. The first category addresses the general capabilities of each model, the second contains specific use cases vital for comprehending dependencies in modern programming languages, and the third concerns the readiness of these models for integration into tools and for conducting further research. While the evaluation criteria may have some subjectivity, we believe they effectively capture the essential functional and non-functional requirements necessary for a comprehensive and practical code structure representation model. A summarized comparison of the results is presented in Table 5.1.

5.2 General model capabilities

For the *general model capabilities* category we specify the following traits:

1. *Language independent* – capability of representing code written in different programming languages. This implies that the analysis of SCG data should function seamlessly irrespective of the program-

	Program execution focus					Software dependency focus				
	CFG	PDG	CPG	SDG	JSysDG	CG	CCN	JRG	GDN	SCG
General model capabilities										
1. Language independent	+	+	+	+	-	+	+	-	+	+
2. Captures interprocedural dependencies	-	-	-	+	+	+	+	+	+	+
3. Captures abstract code entities	-	-	-	-	+	-	+	+	+	+
4. Captures language specific dependencies	-	-	-	-	+	-	-	+	-	+
5. Preserves direct relation to the source code	-	-	-	-	-	-	-	-	-	+
	1	1	1	2	3	2	3	3	3	5
Support for software comprehension use cases										
1. General focus on supporting software comprehension	-	-	-	-	-	-	+	+	+	+
2. Can generate method Call Hierarchy	-	-	-	+	+	+	-	+	+	+
3. Can represent OOP entity relations	-	-	-	-	+	-	+	+	+	+
4. Can be used for software structure analysis	-	-	-	-	+	-	+	+	+	+
5. Suitability for interactive source code visualizations	-	-	-	-	-	-	-	-	-	+
	0	0	0	1	3	1	3	4	4	5
Tool adoption readiness										
1. Extraction program is publicly available for Java language	+	-	+	-	-	+	+	-	-	+
2. The model defines data intermediate representation	-	-	+	-	-	-	-	-	-	+
3. Persistence model is defined	-	-	+	-	-	-	-	-	-	+
4. It is possible to integrate 3rd party analysis tools	-	-	+	-	-	-	-	-	-	+
	1	0	4	0	0	1	1	0	0	4
Readiness for practical software structure comprehension	14%	7%	35%	21%	42%	28%	50%	50%	50%	100%

Table 5.1: Different code graph representations and their capabilities. (CFG = Control Flow Graph; PDG = Program Dependence Graph; CPG = Code Property Graph; SDG = System Dependence Graph; JSysDG = Java System Dependence Graph; CG = Call Graph; CCN = Class Collaboration Network; GDN = General Dependency Network; SCG = Semantic Code Graph)

ming language employed in a project (except in cases where the analysis relies on language-specific extended properties).

2. *Captures interprocedural dependencies* – capability of representing dependencies between different procedures, so the model can be used to analyze the entire project.
3. *Captures abstract code entities* – capability of representing entities which do not directly impact program execution (such as Java interfaces) yet are essential components of code structure.
4. *Captures language specific dependencies* – ability to depict language-specific dependencies (such as inheritance for OOP languages, or *extensions* from the Scala language).
5. *Preserves direct relation to the source code* – ability to represent all entities found in the source code and point to concrete place in the source code where the entity or relation can be found (see Section 3.1).

In this category, we evaluate the model’s ability to comprehend the complete software landscape. Models solely focused on specific monolithic programs, such as CFG, PDG, or related CPG, do not support dependencies between different procedures and also fall short when it comes to handling higher-level dependencies among abstract units. Models that emphasize relationships between code entities prove to be a better fit for software structure comprehension. Most of these models are versatile and can be applied to analyze various programming languages, with the exception of JSysDG and JRG, which are specifically designed for the Java language. It is noteworthy that only SCG ensures a direct connection to the source code through the *location* property. Besides SCG, JSysDG and JRG stand out as the most suitable representations of code, capable of illustrating even language-specific code dependencies within the entire program. Conversely, GDN is constructed on a generic extended concrete syntax tree (eCST), resulting in the loss of specific information, such as distinct node types. For instance, both *enum* and *class* are represented with the same node type referred to as *CONCRETE_UNIT_DECL*.

5.3 Support for software comprehension use cases

To further assess the readiness of the model for software comprehension, we evaluate concrete use cases in the context of the given model:

1. *General focus on supporting software comprehension* – is the model primarily focused on improving software comprehension rather than, for example, prioritizing program optimization.
2. *Can generate method Call Hierarchy* – the ability to generate a call graph and its associated call hierarchy is a widely used representation when understanding software dependencies.
3. *Can represent OOP entities and relations* – can the model effectively represent OOP elements such as objects and classes, as well as relationships such as polymorphism.
4. *Can be used for software structure analysis* – is the model robust and detailed enough to support in-depth analysis for a better understanding of software structures.

5. *Can facilitate source code visualizations* – when examining the graph, can users seamlessly navigate between the graph and the source code? The graph should closely align with the written source code and should be easy to comprehend without introducing unnecessary abstract graph nodes that may clutter the visualization.

Supporting software comprehension is not the primary focus of statement-based program execution models. Consequently, Control Flow Graph (CFG), Program Dependence Graph (PDG), and Code Property Graph (CPG) models are limited to representing single procedures and are inadequate for various defined use cases. The System Dependency Graph (SDG) includes interprocedural call edges, which can assist in illustrating the Call Hierarchy. However, for understanding of software call chains, the Call Graph (CG) is the preferred choice, particularly popular among researchers. JSysDG builds upon the SDG and can accurately represent Object-Oriented Programming (OOP) features for the Java language. Class Collaboration Network (CCN) is also widely employed in software comprehension analysis. This model describes dependencies between classes, which are essential code entities in OOP. CCN is known for its simplicity, producing a manageable number of nodes what facilitates wide range of graph analysis. Models based on code entities offer more direct support for comprehending code dependencies and code structure, making them suitable representations for in-depth software structure analysis. However, interactive source code visualization requires an extremely accurate representation of the source code, complete with information on direct location pointers, a requirement uniquely fulfilled by the Semantic Code Graph (SCG) model.

5.4 Tool adoption readiness

Concerning the readiness for tool adoption, we assessed the following criteria:

1. *Extraction tool is publicly available for Java language* – at least for the Java language (as an established and widely popular language) there is a publicly available tool to generate a representation of a source code.
2. *The model defines data intermediate representation* – a well-designed intermediate representation facilitates adding support for new languages; moreover, analysis tools relying on a common intermediate representation are more versatile and language-agnostic.
3. *Persistence model is defined* – it is inefficient to extract the model from the source code each time an analysis is made. The ability to persist the model is therefore crucial.
4. *It is possible to integrate 3rd party analysis tools* – there should be a means to export the extracted model for analysis with external tools commonly used by researchers, such as Jupyter Notebooks or the Gephi visualization software.

In Chapter 2, we discussed various graph source code models, whose utility on both theoretical and practical levels has been demonstrated by developing experimental extraction tools. Unfortunately, few of these tools are publicly shared, making it challenging for other researchers to reproduce results, or enhance existing implementations, and conduct further analyses. For Java, the Control Flow Graph (CFG) and Call

Graph (CG) can be extracted using the *SootUp*¹ tool [63]. The Code Property Graph (CPG) can be generated using the *joern*² platform, which extracts CPGs for various languages (with stable implementation for C/C++ and experimental for Java/Kotlin/Javascript/Python) into an embedded database and provides a custom Scala DSL for database queries. CPG provides a specification of an intermediate representation³, and it supports the storage of extracted CPG models for future use. Furthermore, it offers seamless integration with third-party software through direct Scala API or export to GraphML, DOT, or other graph formats. The *joern* platform generates Program Dependency Graphs (PDG) and Control Flow Graphs (CFG) for C/C++ languages but it has a potential to support other languages in the future. Shu et al. [106] describe the *JavaPDG* tool capable of extracting Java PDG graphs. However, the link to the tool is outdated, and the actual implementation is currently unavailable. In terms of entity relations, basic networks, such as Class Collaboration Networks, can be extracted using *DependencyFinder*. Extraction programs for PDG, JSysDG, JRG, and GDN for Java language do not currently exist. For the Semantic Code Graph (SCG), we created a command line tool called *scg-cli*⁴. This tool is described in detail in Chapter 7.

5.5 Summary

In summary, the SCG model fulfills all proposed criteria, while GDN, JRG, and CCN achieved the readiness for software comprehension score of 50%. The main shortcomings of these models were related to the lack of direct code relationships, readiness for code visualization, and tool adoption capabilities. Despite its primary focus on comprehensive program execution analysis, JSysDG achieved a score of 42%. Its limitations were mainly attributed to practical constraints within the model and the absence of a comprehensive extraction program. CPG exhibited robust tooling support, primarily due to its extensive utilization in identifying vulnerabilities within C/C++ programs. Although CPG also offers support for the Java language, it remains in an experimental phase, still lacking in terms of feature completeness and model implementation precision. On the other hand, our proposed SCG model maintains excellent software comprehension capabilities with strong practical support for additional tool integration.

¹<https://github.com/soot-oss/SootUp/>

²<https://github.com/joernio/joern>

³<https://cpg.joern.io/>

⁴<https://github.com/VirtusLab/scg-cli>

Chapter 6

Code dependency comprehension through interactive visualization – *Graph Buddy* IDE plugin

To aid the process of understanding code dependencies, we developed *Graph Buddy*, a tool integrated as a plugin for popular IDEs, which enables interactive visual code browsing as a graph of semantic code dependencies. To preserve the direct relation to the source code, visualize, and process only the right amount of details, we utilize the *Semantic Code Graph* (SCG) model (Chapter 3) which is suitable for interactive visualization and capable of representing detailed code semantics.

To evaluate *Graph Buddy*, we conducted a user survey on a group of 10 programmers who used the tool in three programming-related tasks. The survey indicates that the tasks would be difficult to solve without the visual browsing tool. We believe that the proposed solution has a strong potential to enhance the way programmers comprehend code dependencies.

6.1 Motivation and significance

Developers often need to browse through massive source code bases whose code quality tends to degrade over time [12]. Consequently, understanding and maintaining even parts of a large system becomes extremely difficult and time-consuming. There are several places where developers look for information about the system, including documentation, an issue tracking system, or a version control system. However, the ultimate knowledge base is the source code itself. Consequently, we argue that helping the programmer with code understanding increasingly becomes one of the most essential factors in software development.

Analyzing dependencies between modules, classes, methods, and other code definitions is particularly challenging, especially in older projects, where the quality of code architecture often degrades. Various visualizations and visual code browsing improvements can speed up code reading and understanding, and facilitate code maintenance. However, despite the broad academic research in this area, the adoption of code visualization solutions remains low. Designed tools tend to be short-lived prototypes, hard to set up and run, and not integrated within the developers' IDE. Tools also suffer from usability and performance problems

[14]. As such, designed solutions cannot be empirically evaluated in practice in a wide range of projects with comprehensive user feedback. Usability and performance issues are caused not only by implementation shortcomings, but also by inadequate source code abstract representation for a given problem. This often results in low tool responsiveness and cluttered visualizations when used on real projects.

To address the above issues, we developed the *Graph Buddy* tool, a publicly available extension for popular IDEs that allows for interactive visual source code browsing and dependency discovery. Section 6.2 presents related work. Utilization of the SCG in GraphBuddy is described in Section 6.4. We evaluate the tool in Section 6.5. Section 6.6 concludes the chapter.

6.2 Related work

The work described by Bohnet et al. in [17] focuses on visualizing a call graph of methods extracted from C/C++ programs. It requires recording the runtime system interaction for a given feature as well as further measures to extract the relevant part of the caller-callee graph. The user can browse the code interactively by navigating through 3D visualization or by browsing the source code in the “Call Graph Analyzer” program. Alanazi et al. [4] created a tool called *CodEx* that focuses on interactive exploration of call graphs with the help of multilevel hierarchical clustering. Similarly, the work presented by Shah et al. [104] focuses on visualizing call graphs but using interactive microarrays. Tools based entirely on call graphs are missing other important code dependencies and entities, especially those language-specific.

Shu et al. [105] present JavaPDG, a *Platform for Program Dependency Analysis*, which focuses on representing and visualizing various dependencies of code statements, such as System Dependence Graph, Procedure Dependence Graph, Control Flow Graph and Call Graph extracted from Java bytecode. The created tool can help discover program dependencies, but on a very low statement level.

JavaRelationshipGraphs [8] not only focuses on Call Graph, but also takes into account other code relationships (such as inheritance, overrides, ownership). The extracted graph is then visualized with the help of the *Neo4j* graph database. Compared to JavaRelationshipGraphs, SCG can represent all the semantic dependencies of the code and, at the same time, bind them to the exact places in the source code.

Although all of the listed approaches have been proven to be useful, none are widely popularized in daily programmers’ workflows. Moreover, most of the tools presented are not publicly available and lack a broader evaluation from a practical angle of the end users.

6.3 Graph Buddy information model

Our goal is to provide the developer with an efficient visualization to navigate semantic code dependencies between different code units and, in the process, to reduce task cognitive overload and speed up the learning process. For object-oriented programming, that visualization would include supporting call graphs discovery, class inheritance, method overloading, or presenting class overview. To make this possible, we need the right level of information extracted from the source code. Representations such as control flow graphs or data dependency graphs [10], although suitable for various static code analysis and optimizations,

are too granular (statements-level granularity) to visually represent semantic dependencies. Such visualizations look cluttered, too detailed, and hence unreadable to the end user. Moreover, processing unnecessary details results in additional performance and scalability overhead. On the other hand, representations like UML class diagrams are too strict and often lack important properties for code dependency visualization and browsing (such as call graph details).

The Semantic Code Graph, introduced in Chapter 3, has all the required properties to serve as a code representation model for interactive visualization. SCG is a code structure information model that is (a) semantics-rich: it represents a variety of relationships between code definitions; (b) closely tied to the source code; (c) adaptable to various programming languages. As stated in the SCG definition, SCG nodes and edges have to contain the following properties: *location* – location of the related source code; *node/edge type* and *displayName*, a name meaningful for the user. In addition, nodes and edges may contain optional properties depending on their type, e.g., information about the scope of a method or the package containing it. Such a representation allows for modeling information specific for a given programming language with a common representation format constrained only to properties crucial for visualization. Thanks to these properties, SCG is easily translated to interactive graph visualization, e.g. *type* is represented by appropriate node icon, whereas *displayName* is used as node title and *location* allows for highlighting related source code fragments.

6.4 Graph Buddy IDE’s plugins

Graph Buddy aims to speed up the process of analyzing, browsing, and understanding source code dependencies through interactive visualization of the Semantic Code Graph. Graph Buddy is a plugin for Visual Studio Code¹ and IntelliJ² IDEs, publicly available and easily installable through the respective marketplaces. It works out of the box based on generated metadata (see Section 3.4.7) – it is enough to open the Graph Buddy board, and the indexing process will start automatically followed by the loading and introduction screen (Fig. 6.1).

6.4.1 Graph Buddy architecture

The complete Graph Buddy visualization process consists of two phases:

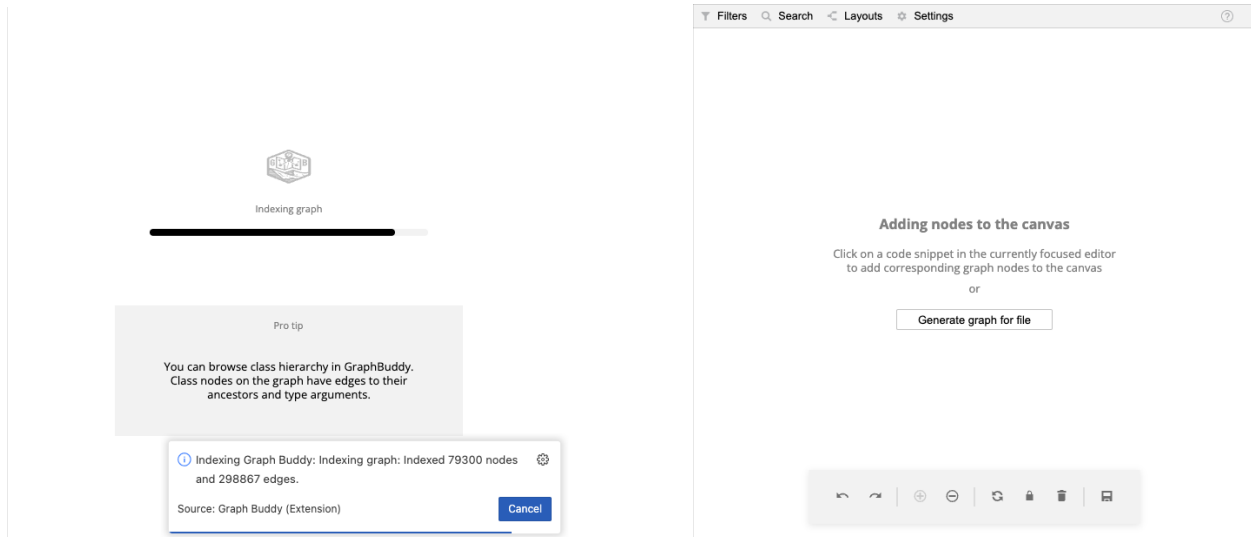
1. Generating graph data in the *.semanticgraph* format during source code static analysis (for Scala with the help of a custom compiler plugin).
2. Indexing and exposing graph data using the Graph Buddy IDE plugin for IntelliJ and Visual Studio Code.

Graph Buddy architecture consists of the following modules (Fig. 6.2):

- Separate Scala and Java code analyzers to generate SCG metadata.

¹<https://marketplace.visualstudio.com/items?itemName=virtuslab.graph-buddy>

²<https://plugins.jetbrains.com/plugin/13467-graph-buddy>



(a) Graph Buddy loading screen. While indexing the data user is presented with useful tips how to use the plugin effectively.

(b) Graph Buddy introduction screen. Initial screen contains a hint how to start using the plugin capabilities.

Figure 6.1: Graph Buddy loading and introduction screen.

- OrientDB embedded graph database to support all queries.
- Graph Buddy Server to support operations needed for visualizations and visual code browsing.
- Graph Buddy Visualization library to provide extracted standard visualization capabilities for both VS Code and IntelliJ plugins.
- VS Code and IntelliJ plugins to natively extend IDE's capabilities with Graph Buddy functionality.

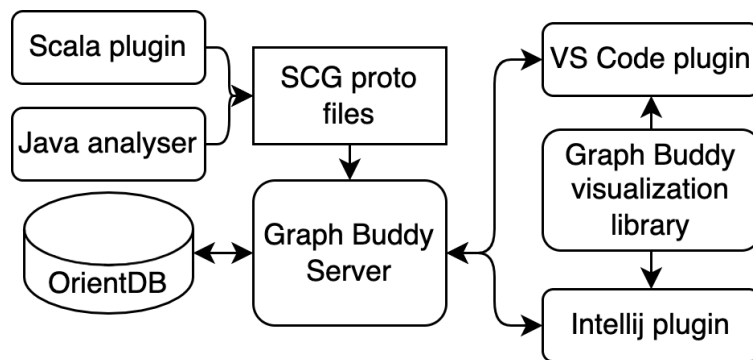


Figure 6.2: Graph Buddy architecture – SCG protobuf files are the source information model for visualization.

6.4.2 Graph Buddy user interface

The Graph Buddy integrates closely with the IDE source code editor. Each definition marked by the cursor in the source code file is drawn on the interactive Graph Buddy board (Fig. 6.3). Furthermore, by clicking

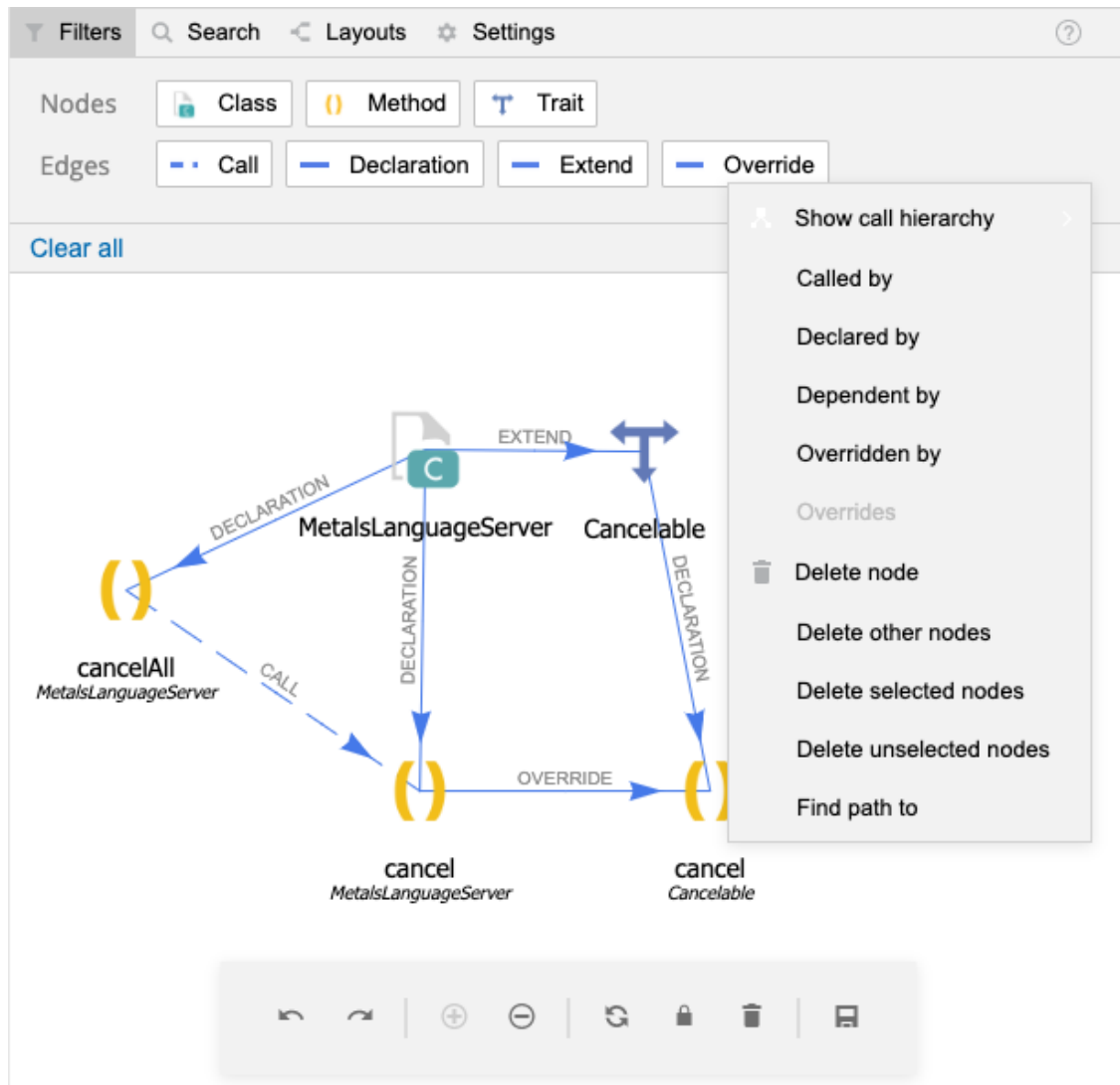


Figure 6.3: Graph Buddy user interface with partial graph visualization of **MetalsLanguageServer** class – its two declared methods, *cancelAll* directly calling *cancel*, and method *cancel* overriding method defined in *Cancelable* trait extended by *MetalsLanguageServer*. The user can interactively inspect the related source code by selecting the node or edge, analyze additional dependencies through a context menu, or filter nodes or edges by their type.

on a graph node or edge, the appropriate place in the source code file will be highlighted. From the context menu, it is possible to display the call hierarchy (with a chosen depth of recursive neighbors), find a path between nodes, or find only specific dependencies. Users can also filter out some types of nodes or edges, choose a predefined layout (classical, horizontal or vertical) or adjust it manually, delete or search through the displayed nodes. Visualizations can be stored in a separate file for later use or sharing with other users.

Changes to the source code require regenerating related SCG meta files by running regeneration process (Java), or by recompiling the project (possible incremental compilation) (Scala), and refreshing the server.

6.5 Graph Buddy evaluation

6.5.1 Graph Buddy performance

To validate the performance of Graph Buddy, we configured and compiled one of the largest and most popular Scala open source projects, the Akka library³. Akka contains 8,120 files and 243,193 lines of code. The Semantic Code Graph generated for this project has 260,607 nodes and 796,939 edges. It took 60 seconds for the plugin to load the Semantic Code Graph data on the server (macOS, 2.3 GHz Intel Core i7 with 32 GB RAM) with a maximum memory usage of 2GB. The plugin was perfectly responsive despite the significant number of nodes in the graph. This proves that the method is scalable to large code bases.

6.5.2 Graph Buddy user feedback loop

Graph Buddy users can raise the feedback through score and comments⁴ on the plugin's marketplace. At the time of writing, the VS Code plugin was downloaded more than 3.8k times and has the score 4/5, IntelliJ plugin was downloaded about 6.8k times with the score 4.2/5. The user can also post issues and seek help through the official Github project repository.⁵ If the user agrees, the plugin will also collect anonymous events generated by Graph Buddy. Analyzing these events gives information on active users, the most likely used features, or potential usability problems. Establishing a short user feedback loop, although often overlooked by researchers, is a great opportunity to evaluate the usefulness of the tool and introduce the necessary changes.

6.5.3 Graph Buddy usability user survey

In addition to the existing user feedback loop, to validate the usefulness of our solution, we prepared the following experiment using the Graph Buddy plugin. We asked ten programmers (volunteers within VirtusLab with professional experience from 2 to 10 years) to perform three programming-related tasks on the *metals* repository⁶ – which was chosen as a typical medium-sized open source product repository, a close example

³<https://github.com/akka/akka>

⁴For example, in this comment <https://plugins.jetbrains.com/plugin/13467-graph-buddy/reviews#review=71900> a user praises the usefulness of Graph Buddy in IntelliJ marketplace: "Absolutely incredible, thank you so much! As a visual person, this helps me «follow» the line of a code path so much easier than ctrl+clicking and trying to remember everything."

⁵<https://github.com/VirtusLab/graphbuddy>

⁶<https://scalameta.org/metals>

to what the user might encounter in day to day work – using the Graph Buddy plugin inside their favorite IDE:

1. **T1 Finding a call hierarchy.** An *Endpoint* in the *metals* project is a method annotated with `@JsonRequest`. Find all the endpoints that indirectly call the `ReferenceProvider#sortByLocationPosition` method. **Rationale.** Analyzing the call hierarchy of a given method allows assessing which parts of the system will be affected after changing the method implementation.
2. **T2 Finding a call path.** Find the shortest call path between `MetalsLanguageServer#rename` and `SymbolAlternatives#isVarSetter`. **Rationale.** Analyzing call path can quickly reveal whether a given place in the code can be affected by a given internal method.
3. **T3 Learning about the project from Graph Buddy visualization.** Load a visualization saved in *metals-graph.json* file (Fig. 6.4) into Graph Buddy. Learn about *metals* code actions and describe the knowledge you can gain from this visualization. **Rationale.** Programmers often need to explain the dependencies of the source code to the team member unfamiliar with the project.

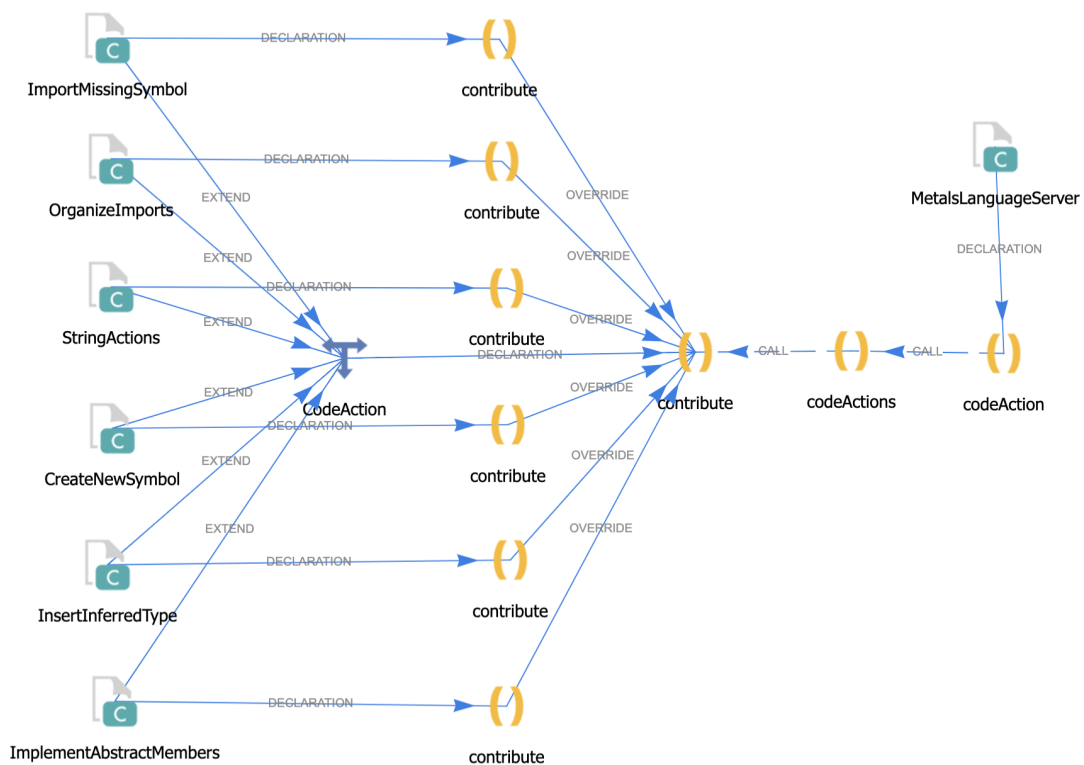


Figure 6.4: The dependencies of available code actions in the *metals* library presented in Graph Buddy

Tasks **T1** and **T2** represent common and important *reachability questions* problems [64]. For task **T1** the preferred solution (Fig. 6.5) using the Graph Buddy tool required first localizing the definition of `ReferenceProvider#sortByLocationPosition` in the source code and then displaying the call graph

hierarchy with the Graph Buddy "Show call graph hierarchy" context menu option. As a result, the user gets an interactive call hierarchy graph with endpoints marked with blue dots (methods which are not called directly in the code) – *definition*, *references*, and *rename*. The user can also follow a particular call path easily – selecting the node highlights its definition in the source code, and selecting the edge highlights the method invocation. By contrast, alternatively the user can use the *show call hierarchy* feature built in IDEs (both in IntelliJ and VS Code). The results are then presented as a flat expandable tree of all call paths (Fig. 6.5). However, this representation has significant drawbacks:

- it is hard to localize unique endpoints as endpoints are displayed multiple times, separately for each branched call path,
- it is hard to follow particular call graph as each line represents only call edges, and finding node definitions requires manual code browsing,
- it is hard to spot the additional properties of the code graph otherwise clearly visible in Graph Buddy representation (e.g., the **rename** method having three different possible call paths to **sortByLocationPosition**).



Figure 6.5: Graph Buddy (left) vs IntelliJ's call graph solution to task **T1**.

Task **T2** focuses on finding a possible path between two nodes. With Graph Buddy, it was easily possible to find both nodes and use the "Find path to" feature from the context menu. IDEs, on the other hand, do not have a similar functionality (as of writing this article); the best option⁷ is to use again the *Call Hierarchy* feature and manually follow each path, starting from the **SymbolAlternatives#isVarSetter** method, searching for the **MetalsLanguageServer#rename** endpoint (Fig. 6.6). The main drawback of this solution is that the process is prone to errors and slow. In the worst-case scenario, when the path does not exist, it

⁷<https://stackoverflow.com/questions/34841440/intellij-find-code-path-between-two-methods>

Q	Yes	No	Success rate
Q1	8	2	80%
Q2	10	0	100%
Q3	9	1	90%

Table 6.1: Graph Buddy evaluation survey’s results

requires manually going through each possible call graph from the starting method, which in bigger projects might be impossible to complete in a reasonable time.

In task **T3**, the user had to browse through prepared visualization of the code. It turned out that the user was able to quickly list all the implemented actions, spot the common **CodeAction** trait with **contribute** method, read the actual implementation and localize the main method usage. It proves that proper interactive visualization can describe parts of the system efficiently, without the need for additional verbal explanation.



Figure 6.6: Graph Buddy (left) vs IntelliJ find a path solution to task **T2**. The text is highlighted artificially to mark the possible call path between the *rename* and *isVarSetter* methods.

After completing these three tasks, we prepared a short survey:

- **Q1** Did Graph Buddy help you find and understand the call hierarchy?
- **Q2** Did Graph Buddy help you find and understand the call path between two methods?
- **Q3** Were you able to complete the tasks?

The survey results, presented in Table 6.1, strongly indicate that the users found Graph Buddy helpful in solving tasks **T1** and **T2**. Analysis of the graph in task **T3** allowed participants to easily comprehend the structure of the code of *metals* actions. Furthermore, all participants agreed that tasks **T1** and **T2** would be difficult to solve without the Graph Buddy plugin.

6.6 Conclusions and future work

In this Chapter, we demonstrated Graph Buddy – an interactive code browsing tool integrated with popular IDEs built on top of the Semantic Code Graph information model for representing detailed source code semantic dependencies. The plugins are publicly available with open channels for direct user feedback. Our initial usefulness evaluation proved that with Graph Buddy it is possible to browse and discover vital code dependencies in a more efficient way than with standard IDE features, even on large projects. In the future, we want to focus on user feedback and analyze available metrics to evaluate possible directions to enable massive plugin adoption.

Chapter 7

scg-cli command line tool to facilitate software comprehension

In this chapter, we present *scg-cli*, a command line tool facilitating software comprehension. The tool extracts semantic information about code structure and dependencies from the Java and Scala projects, and structures it as a Semantic Code Graph (SCG), an information model underlying *scg-cli*. The SCG data, once written into a portable, open protobuf-based format, can be used by the *scg-cli* command line tool to obtain project metrics, find the most critical code entities, and compute project partitionings. The results of this analysis and the SCG data can be exported for further investigation by external tools such as Gephi software (visualization) and, notably, as a Jupyter Notebook environment with helper APIs to enable advanced analysis of the project using data analytics methods. We explain functionalities of the *scg-cli* tool and demonstrate its capabilities by showing an example analysis of an open-source Java project *commons-io*.

7.1 Motivation and significance

Every day, programmers write new lines of code, fueling the growth of software repositories. Once written, the code undergoes multiple rounds of reading and modification, often not by the original author. As a result, software maintainers are faced with the task of comprehending unfamiliar sections of the software [2]. In addition, developers widely reuse open-source libraries, which often entails the need for a deep understanding of the library's source code, not only to enable proper usage but also to introduce bug fixes and new features.

Understanding unfamiliar software, whether it is an in-house project or a third-party open-source library, can be an arduous and time-consuming undertaking for developers [123]. They strive to gain comprehensive knowledge of the project as quickly as possible, however, while project documentation typically offers instructions on building and preparing merge requests, there is often a lack of architecture documentation that provides a holistic view and introduction to the project. Consequently, critical information about the software needs to be extracted directly from the source code. The *scg-cli* aims to accelerate the software comprehension [103, 107] process by providing insights extracted directly from the source code, expediting the learning curve for developers.

The *scg-cli* utilizes the Semantic Code Graph (SCG), an information model closely tied to the source code, to capture software artifacts and code dependencies. By leveraging this model, the *scg-cli* can generate SCG data from Java projects and employ various software comprehension functionalities based on this data. Additionally, it offers the potential for extending analysis to other languages either through enhancing the capabilities of *scg-cli* or by utilizing external SCG data extractors, such as the separate compiler plugin available for Scala¹. To ensure compatibility, integrations for other languages must adhere to the official metadata structure defined in the protobuf format².

The *scg-cli* command line tool enables software comprehension of projects in several aspects. Currently, its capabilities include:

- Generating a project summary to give the programmer high level overview of the project.
- Finding critical entities in the source code that are usually good starting points for reading the project.
- Discovering and suggesting software partitioning to expose module boundaries or support refactoring efforts.
- Exporting SCG data and analysis results to graph formats (.GDF, graphml) for external visualizations.
- Providing a Jupyter-based environment for further software mining through analysis of SCG data using popular scientific libraries, such as pandas³, or networkx⁴.

The functionalities of the *scg-cli* significantly reduce the time needed to comprehend software by extracting and presenting invaluable project-specific knowledge. Furthermore, it establishes a robust foundation for future functionalities, tools, and extensions developed based on the SCG model of the source code.

7.2 Related work

In software development, the activities of software comprehension and maintenance are typically performed within integrated development environments (IDEs). IDEs primarily enable efficient browsing and manipulation of source code files. Browsing the source code is a key element in software comprehension, and within IDEs, it is augmented with syntax highlighting, go-to capabilities, file hierarchy, and multiple other facilitators. However, IDE-based software comprehension is focused on the particular context with a limited scope of the source code unit. For advanced and comprehensive software analysis there exist dedicated tools such as *JArchitect*⁵, or specialized IDE plugins, such as *stan4j*⁶. *JArchitect* is a standalone tool capable of monitoring code quality, comparing releases, or finding and prioritizing technical debt for refactoring activities. It also provides project overview through exploring software architecture capabilities or trends monitoring. However, its software comprehension capabilities are quite limited, e.g., it lacks the ability to

¹<https://github.com/VirtusLab/scg-scala>

²https://github.com/VirtusLab/graphbuddy/blob/master/proto/graph_node.proto

³<https://pandas.pydata.org/>

⁴<https://networkx.org/>

⁵<https://www.jarchitect.com/>

⁶<http://stan4j.com/intro/>

find crucial code entities. Moreover, the tool is not open source and hence impossible to extend, it is restricted to a paid license and only supports the Java language. *stan4j* analyzes code structure quality based on the generated Java bytecode. It has an Eclipse integration plugin to enable code structure visualization and browsing, which can help with software comprehension capabilities. On the other hand, it is a paid tool for bigger projects (the free version is limited to 500 classes), and does not allow for custom analysis. The tool is also restricted to Java support and it has not been actively maintained since 2021. There are multiple widely used tools available for static code analysis focusing on code review, security or code quality, such as *ScalaStyle*⁷, *Checkstyle*⁸, or monitoring platforms such as *SonarQube*⁹ or *Fortify*¹⁰. These tools, however, are not suitable for supporting the software comprehension process. In contrast, *scg-cli* focuses on software comprehension in the context of the entire project, possibly narrowing down the investigation scope if needed.

7.3 Software description

7.3.1 Software architecture

The *scg-cli* is a command line application written in Scala 3 and it can be built directly from the sources using the `sbt universal:packageBin` command, but for user convenience binary artifacts are distributed through the official GitHub page¹¹. A compressed package contains compiled source code with run scripts for MacOS, Linux and Windows operating systems. For basic usage, the tool requires only a Java runtime preinstalled on the host machine.

The architecture diagram for the *scg-cli* tool is presented in Fig. 7.1. All interactions with the tool are invoked by user commands and processed by the CLI processor. Three different modules can be involved in the process of executing user commands:

- load/generate data module – responsible for initial source code analysis and generating SCG protobuf metadata files or loading the SCG data,
- project analysis module – conducting various project analyzes based on the SCG information model,
- export module – exporting SCG data to other formats (e.g. GDF, .graphml) for further analysis with external tools such *Gephi*¹² or exporting and running a Jupyter notebook¹³ with loaded SCG data.

The capabilities of *scg-cli* can be leveraged by external software analysis tools by utilizing directly its command line interface. This integration pattern is widely embraced in the industry for interacting with third-party software.

⁷<http://www.scalastyle.org/>

⁸<https://checkstyle.org/>

⁹<https://sonarqube.org>

¹⁰<https://www.microfocus.com/en-us/cyberres/application-security>

¹¹<https://github.com/VirtusLab/scg-cli/releases>

¹²<https://gephi.org/>

¹³<https://jupyter.org/>

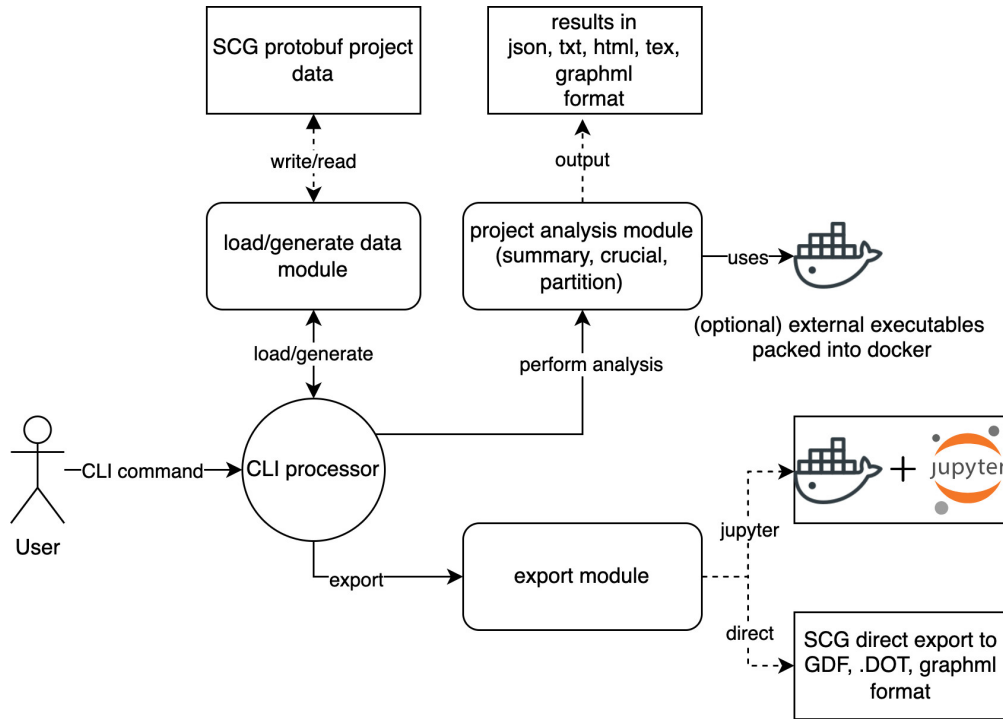


Figure 7.1: The *scg-cli* software architecture. The *load/generate data module* generates SCG data files for a project. The *project analysis module* performs analysis of the project using the SCG data. The *export module* enables custom analysis using external tools known to researchers, such as Jupyter.

7.3.2 Software functionalities

The *scg-cli* implements helper command **scg-cli help** which summarizes software functionalities, as presented below.

```

Usage: scg-cli [COMMAND]
CLI to analyze projects based on SCG data
Commands:
  help      Display help information about the specified
            command.
  crucial   Find crucial code entities.
  generate   Generate SCG metadata.
  partition Suggest project partitioning.
  summary   Summarize the project.
  export     Export SCG metadata to various output formats.

```

Brief information about the usage of each particular command can be obtained with **scg-cli help [COMMAND]**. In the following subsections we provide a detailed explanation of the major *scg-cli* capabilities.

Generating SCG data

The *scg-cli* tool can generate SCG protobuf data for Java projects using the command **scg-cli generate -l java <workspace>**. The generated data is stored within the project structure in the

.semanticgraphs folder. Additionally, it is possible to generate SCG data for Scala projects using the Scala plugin¹⁴. These files can be later loaded into the *scg-cli* for further analysis.

Presenting project overview

Project overview allows one to grasp the size of the project and other interesting characteristics. The **scg-cli summary <workspace>** command produces a short project overview containing SCG metrics, such as the number of nodes and edges and their distribution by type, total number of code lines, graph density, average node in- and out-degree, global clustering coefficient, degree assortativity coefficient. Each metric provides some insight about the project, e.g.:

- Number of nodes and edges – a simple metric to assess the project size and monitor its growth over the time.
- Density – high SCG density can indicate higher code dependency coupling and general complexity.
- Average Degree – average degree of nodes in the project; large average degree can indicate high project coupling.
- Global Clustering Coefficient – high value can indicate well-modularized software with high module cohesion and low coupling.
- Node type distribution – provides various insights, e.g., a high number of variables can entail harder reasoning about the code and overall lower software quality.

Finding crucial code entities

The most important central code entities are typically good starting points for reading the source code in the process of software comprehension [31]. The *scg-cli* automatically finds the most important nodes in the SCG graph based on the set of nine different metrics [43], including LOC (line of codes associated with a code entity), out-degree and in-degree, eigenvector centrality, Katz centrality, PageRank, betweenness centrality and harmonic centrality. Each metric represents the importance of a node from a different perspective. We also introduced an additional metric – *Combined importance* – which represents node importance aggregating all other metrics, i.e., a node with high score in multiple metrics will also have a high combined importance.

Suggesting project partitioning

Software Module Clustering is a data mining technique that groups software entities into clusters sharing the same features [99], facilitating software comprehension [78] or refactoring activities, e.g., software modularization. The *scg-cli* tool provides clustering functionality which uses graph partitioning algorithms [26] to divide the SCG graph into a certain number of partitions while minimizing, e.g., the number of edges that

¹⁴<https://central.sonatype.com/namespaces/org.virtuslab.semanticgraphs>

cross partitions. The maximum number of the partitions is provided as a command line argument *n*:

```
scg-cli partition -o {csv,json,html,txt,tex,graphml} <workspace> {n}.
```

As the SCG graph often contains thousands of nodes, the *scg-cli* tool uses two efficient graph partitioning algorithms implemented in the *gpmets* [57] and *patoh* [28] software. Both algorithms compute partitions within seconds even for large graphs.

The partitioning functionality will not only present the node allocation to each partition but also will compare the quality and characteristics between different partitionings. The following properties of the partitioned graphs are compared:

- Modularity – the ratio of edges between nodes inside a partition to edges crossing the partition boundary.
- Edge cut size – how many edges goes between different graph partitions.
- Weighted and Average Accuracy by file and by package unit – average percentage of nodes within a given file or package unit which belong to the same partition. For a well structured software all nodes in a particular file or package should be assigned by the algorithm to the same partition. Outstanding nodes might imply the need to reorganize the source code and improve the project structure quality. Weighted Accuracy additionally takes into account the size (in term of the total number of nodes) of a particular file or package.
- Partition variance – a measure of partition size similarity. A value close to zero means that the number of nodes in all partitions are equal, while a value close to 1 denotes highly unbalanced partitions in terms of size.
- Nodes distribution – a summary which presents the number of nodes within each partition (as percentage of all nodes).

Exporting for further analysis in other tools

The *scg-cli* can export the data to external format, i.e. GDF¹⁵, .DOT¹⁶ or graphml¹⁷ which can be later consumed in other tools such as Gephi¹⁸ or Graphviz¹⁹. Moreover, a special *jupyter* export allows to run a Jupyter notebook in a Docker container and load all SCG data into it for further software mining activities, using command **scg-cli export -o jupyter <workspace>**. The command mounts the SCG data as a Docker volume and starts a publicly available Docker image²⁰ containing a Jupyter notebook along with a helper API to load and operate over SCG data. After successful container startup the resulting local Jupyter notebook URL is presented to the console.

¹⁵<https://gephi.org/users/supported-graph-formats/gdf-format/>

¹⁶[https://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language))

¹⁷<https://en.wikipedia.org/wiki/GraphML>

¹⁸<https://gephi.org/>

¹⁹<https://graphviz.org/>

²⁰<https://hub.docker.com/r/liosedhel/scg-jupyter>

Extracting and analyzing CCN and CG networks with *scg-cli*

The *scg-cli* also has the capability to extract and operate on two additional code graph models discussed in Chapter 5: the Class Collaboration Network (CCN) and the Call Graph (CG). These models are extracted from the corresponding SCG data. The *scg-cli* command can be run with the `-g CCN, CG` argument to switch the analysis to a specific network type. We also plan to add support for other types of software networks in the future, such as Package or File Collaboration Networks. Notably, *scg-cli* can operate over any network type as long as it follows the SCG protobuf format, outlined in the *scg-cli* documentation²¹.

7.4 Illustrative Examples

In this section, we demonstrate several software analyzes conducted with the *scg-cli* tool, using the *commons-io*²² project as an example. The *commons-io* project is an open-source library widely used in the industry to facilitate various file and stream related functionalities²³ and it is a part of commonly known Apache Commons set of libraries²⁴.

7.4.1 Generating SCG data

To begin the analysis, the project sources have to be downloaded to the local disk. The *scg-cli* will analyze local files and generate the SCG data using command `scg-cli generate -l java commons-io`. The program will parse source files and run detailed analysis to extract a complete SCG model for all Java sources and store resulting protobuf files into the `.semanticgraphs` folder. All further analysis using *scg-cli* are performed based on the generated metadata without the need to analyze the source code files again.

7.4.2 Project overview

To generate the project overview in the HTML format, one needs to run `scg-cli summary -o html commons-io`. The result, presented in Fig. 7.2, contains computed graph parameters and a chart with distribution of node and edge types. The parameters are explained in the presented table.

7.4.3 Finding crucial nodes

Finding crucial nodes is a *scg-cli* feature that analyzes the entire project and applies different scoring metrics to assess importance of graph nodes (i.e., code entities) from different perspectives (the different metrics are described in Section 7.3.2). Table 7.1 presents the top 3 nodes in each category computed for the *commons-io* project, obtained from the output of command `scg-cli crucial -n 3 -o tex commons-io`. Looking at the combined importance metric, the most important nodes are `IOUtils` (utility class which enhances

²¹<https://github.com/VirtusLab/scg-cli/blob/main/src/main/protobuf/graphnode.proto>

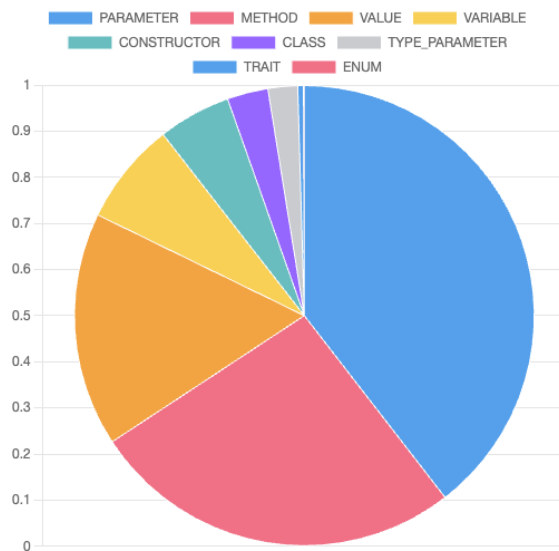
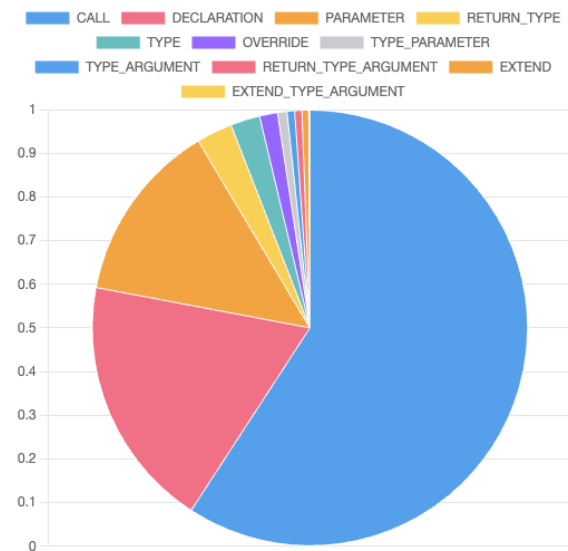
²²<https://github.com/apache/commons-io>

²³<https://commons.apache.org/proper/commons-io/>

²⁴<https://commons.apache.org/>

commons-io-2.12.10 project summary

Parameter	Value	Description
Nodes	8261	Number of nodes in the graph (such as classes, methods, variables, etc.).
Edges	24257	Number of edges in the graph. See the chart below for distribution.
Total LOC	46526	Total number of lines of code.
Density	0.00035549	Density of the graph -- a value between 0 (low density) and 1 (high density). High SCG density can indicate higher code dependency complexity.
Average Degree	2.936	Average number of edges. High average degree of nodes, especially when combined with high density, suggests high project coupling.
Global Clustering Coefficient	0.07019667	High value indicates that the graph tends to create interconnected clusters, low value indicates the network being sparse or fragmented. For SCG, high GCC value can indicate well-modularized software with high module cohesion and low coupling.
Degree Assortativity Coefficient	-0.23721884	Positive value indicates that nodes with the same degree tends to be connected, negative value suggests mostly connections between nodes of different degree, value close to zero suggests random network. For SCG, low DAC can suggest tendency of the graph towards clustering and establishing hubs (connections between nodes with different degrees).

Node distribution**Edge distribution**Figure 7.2: Project overview for the *commons-io* project generated with the **scg-cli summary** command.

operations on streams) and **FileUtils** (utility class which enhances operations on files). These classes indeed contain the most desirable and useful *commons-io* library functionalities. *IOFileFilter* is a trait with 31 different implementations facilitating file filtering, complementing **FileUtils** functionalities.

Lines of code	Score
o.a.c.i.IOUtils	3608
o.a.c.i.FileUtils	3434
o.a.c.i.file.PathUtils	1682
Node out-degree	Score
o.a.c.i.FileUtils	176
o.a.c.i.IOUtils	171
o.a.c.i.file.PathUtils	106
Node in-degree	Score
o.a.c.i.filefilter.IOFileFilter	131
o.a.c.i.CloseableURLConnection?urlConnection	47
o.a.c.i.function.IOBaseStream.unwrap()	46
PageRank	Score
o.a.c.i.filefilter.IOFileFilter	0.013
o.a.c.i.output.NullPrintStream	0.011
o.a.c.i.output.NullPrintStream.NullPrintStream()	0.006
Eigenvector centrality	Score
o.a.c.i.function.IOBaseStream.unwrap()	0.307
o.a.c.i.input.Tailer	0.272
o.a.c.i.function.IOStream.T	0.201
Katz centrality	Score
o.a.c.i.filefilter.IOFileFilter	2.340
o.a.c.i.CloseableURLConnection?urlConnection	1.476
o.a.c.i.function.IOBaseStream.unwrap()	1.467
Betweenness centrality	Score
o.a.c.i.function.IOStreams.forAll()	301648.755
o.a.c.i.function.IOConsumer	299069.829
o.a.c.i.function.IOStream.adapt()	272351.667
Harmonic centrality	Score
o.a.c.i.FileUtils	0.092
o.a.c.i.IOUtils	0.070
o.a.c.i.FileUtils.FileUtils()	0.064
Combined importance	Score
o.a.c.i.FileUtils	3
o.a.c.i.filefilter.IOFileFilter	3
o.a.c.i.IOUtils	3

Table 7.1: Importance analysis: top 3 most important nodes (code entities) in the *commons-io* library computed based on 9 different metrics.

7.4.4 Project partitioning

The *common-io* project consists of just one module. If we were to divide this software into smaller, dedicated libraries, we would need to consider the number of libraries and their boundaries. Partitioning the project can also unveil unique software functionalities or identify whole subsystems, which supports the process of software comprehension. The *scg-cli* tool can suggest partitionings of the project graph and assess their quality. For example, command `scg-cli -o tex partition commons-io 10` computes partitionings of the *commons-io* project, generating from 2 up to 10 partitions. The user will be presented with the partitioning summary containing information shown in Table 7.2. From these results we can deduce that the proposed project split into 6 partitions (using the gpmetis algorithm) presents high quality, despite of significant number of partitions, as measured by modularity metric equal to 158.9, 45 edge cut size and very high accuracy (for file weighted and standard accuracy – 95% and 97% respectively and package weighted accuracy equal to 70%). The accuracy is higher than in the splits to 4 or 5 partitions, and the modularity is only slightly lower (only 4 more edges to cut). The sizes of the partitions are not evenly distributed with variance above 0.644 and small first and fourth partitions. To look at details of this split, we can export a partition summary in the HTML format: `scg-cli -o html partition commons-io 10`. For each partition split, the partition summary presents the dominant partition number for each package and file. Exporting to CSV or graphml is also possible which enables even further analysis, as shown in the next section.

Method	N	Modularity	Cut Size	FWA %	FA %	PWA %	PA %	CV	Distribution %
gpmetis	2	3597.0	2	99	99	99	99	0.932	[3,96]
patoh	2	1798.0	4	99	99	90	93	0.852	[7,92]
gpmetis	3	311.9	23	96	99	71	86	0.669	[2,41,55]
patoh	3	286.8	25	97	99	75	88	0.615	[8,58,32]
gpmetis	4	140.1	51	94	96	63	80	0.538	[8,36,15,40]
patoh	4	166.3	43	95	97	65	82	0.328	[35,23,28,12]
gpmetis	5	174.5	41	95	96	66	81	0.692	[9,15,2,38,33]
patoh	5	113.2	63	92	97	59	80	0.232	[20,20,11,25,21]
gpmetis	6	158.9	45	95	97	70	85	0.644	[2,15,30,3,22,26]
patoh	6	118.9	60	91	96	53	75	0.426	[10,29,18,7,17,15]
gpmetis	7	104.8	68	88	94	53	77	0.449	[3,20,17,10,7,19,20]
patoh	7	103.3	69	91	94	55	75	0.251	[7,18,14,13,10,17,17]
gpmetis	8	100.4	71	92	96	58	78	0.500	[7,21,8,4,18,7,20,12]
patoh	8	87.8	81	90	93	50	74	0.276	[17,18,13,10,8,12,11,7]
gpmetis	9	75.6	94	88	94	53	76	0.464	[4,11,8,17,11,8,3,17,17]
patoh	9	74.7	95	87	93	52	75	0.276	[10,7,11,10,7,10,8,14,17]
gpmetis	10	85.7	83	88	93	50	73	0.493	[4,15,4,10,17,8,7,3,13,15]
patoh	10	80.8	88	89	93	48	73	0.285	[10,7,8,15,13,5,7,7,12,11]

Table 7.2: Different partitionings of the *commons-io* library and their quality computed by *scg-cli*.

7.4.5 Exporting and analyzing in external tools

The *scg-cli* tool can export the SCG data for further analysis in Jupyter with the **`scg-cli export -o jupyter commons-io`** command. For instance, assume we want to analyze the high quality partitioning of the *commons-io* library into six partitions, obtained with *gpmetis* program. More specifically, we would like to find methods that belong to a different partition than their parent class. Classes with such methods can be problematic when splitting the project since they are probably used extensively by different parts of the project. Listing 7.1 shows the code that performs this analysis. First, we load the CSV output of the partitioning into six partitions into the notebook (line 8) and merge the results into one data frame (line 6). For each method (line 22) we add two additional columns: *parent* which is the identifier of the parent node, and *parent_npart* which is the partition number of the parent (line 23). Then we find the methods whose partition number is not the same as the one of their parent classes (line 25) and group the results by parent identifier to find the classes with the largest number of such methods. Looking at the results, we find that classes *FileUtils* and *IOUtils* have, respectively, 14 and 7 methods that belong to different partitions. Consequently, these classes contain functionalities utilized by different partitions and splitting the software into separate modules will likely require some prior code movements and refactorings.

Listing 7.1: Finding methods assigned to different partitions than their parent class

```

1  import pandas as pd
2  import scg
3
4  scg_files = scg.read_scg("commons-io")
5  G = scg.create_graph(scg_files)
6  scg_df = pd.merge(
7      scg.create_nodes_df(scg_files),
8      pd.read_csv("commons-io-npart-6-gpmetis.csv"),
9      on="id",
10 )
11 def find_parent_and_parent_npart(m):
12     parent = next(
13         p
14         for p, _, data in G.in_edges(m["id"], data=True)
15         if data["type"] == "DECLARATION"
16     )
17     p = scg_df.query("id==_@parent")
18     return pd.concat(
19         [p["id"], p["npart"]], ignore_index=True
20     )
21 m_df = scg_df.query('kind=="METHOD").copy()
22 m_df[["parent", "parent_npart"]] = m_df.apply(
23     find_parent_and_parent_npart, axis=1
24 )
25 m_df[m_df["npart"] != m_df["parent_npart"]].groupby(
26     "parent"
27 ).size()
28 #o.a.c.io.ByteOrderMark
```

1

29	<code>#o.a.c.io.FileUtils</code>	14
30	<code>#o.a.c.io.IOUtils</code>	7
31	<code>#o.a.c.io.comparator.LastModifiedFileComparator</code>	1
32	<code>#o.a.c.io.file.PathFilter</code>	1
33	<code>#o.a.c.io.file.PathUtils</code>	2
34	<code>#o.a.c.io.filefilter.EmptyFileFilter</code>	1
35	<code>#o.a.c.io.input.BOMInputStream</code>	1

We can visualize the results of the partitioning with the Gephi tool, by exporting the partitioning results as a graph in the graphml format, with additional node properties concerning the partition number in a given split: `scg-cli -o gml partition commons-io 10`. The node colors presented in Fig. 7.3 denote the assigned partitions.

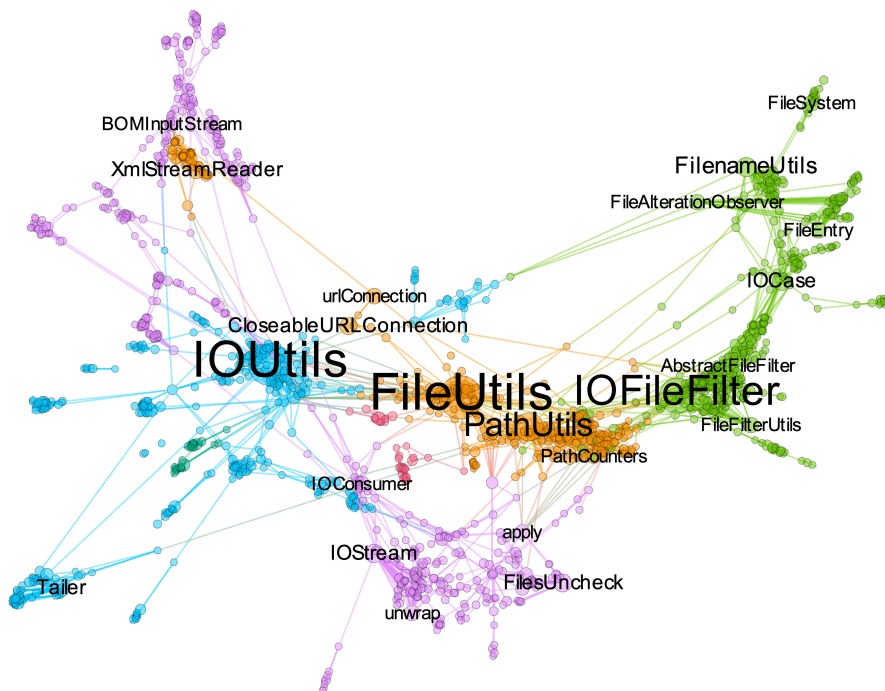


Figure 7.3: Top nodes for the *commons-io* project. Colors denote partitions (modules) obtained from 6-partition split using the *gpmets* algorithm. Visualized with Gephi.

7.5 Impact

With the enormous amount of new code created every day, enhancing software comprehension speed is increasingly crucial. The *scg-cli* tool, with the underlying theoretical concept of the Semantic Code Graph, enables comprehensive project analysis focused on understanding the code structure. The tool allows for quick insight into the project using a simple command line interface. Presenting project overview, finding critical code entities, and suggesting software partitioning (discovering modules) gives the programmer a good basic understanding into the project.

However, the capabilities of the tool do not end there. The semantic data describing the code structure is written in an open intermediate protobuf format, easy to consume by other tools for more in-depth analy-

sis. To our best knowledge, no existing tool has this capability. Consequently, our approach can potentially impact future tools focusing on code dependency and structure analysis, and overall software comprehension. The *scg-cli* tool itself provides functionality to export and load the SCG data into the Jupyter notebook, enabling a split-apply-combine style of analysis using dataframes, a well-established data analytics approach [120].

The *scg-cli* is open sourced on the GitHub platform allowing further collaborative development of this project. Other researchers can contribute by adding SCG data generation capabilities for new languages, extending currently available set of functionalities, or adding more scripts into Jupyter notebooks to enable more detailed analysis. Hopefully, this software will also have a positive effect towards new generation of development tools focused on supporting software comprehension.

7.6 Conclusions

In this chapter we presented *scg-cli*, a command line tool supporting software comprehension of Java and Scala projects. The process of analysis begins with extracting semantic information from the project source code in the form of the Semantic Code Graph, written in an open protobuf-based format. The *scg-cli* provides insight into project structure and basic metrics, finds crucial software entities, and helps discover modules by computing graph partitionings. More in-depth analysis can be performed using the export functionality, which includes a Jupyter notebook environment.

In the future we would like to add support for additional languages, implement new functionalities, and build a community around the *scg-cli* GitHub project. This will allow us to get the feedback and the support to extend tool capabilities.

Chapter 8

Empirical software comprehension with Semantic Code Graph

8.1 Introduction

In this Chapter, we perform various experimental software comprehension activities on the eleven open source projects described in Section 4.3. We extract three graph models – the SCG, the CCN, and the CG – and analyze the gathered data using different techniques, such as complex network analysis [81] and the split-apply-combine data analysis methods [120]. We have conducted the experiments with the help of the *scg-cli* tool described in Chapter 7. From a practitioner’s standpoint, our objective is to provide developers with an comprehensive introduction to the analyzed open source projects and showcase how similar analysis can be conducted over any Java or Scala software.

For each project, we build and analyze the full SCG graph, and then extract the Class Collaboration Network and the Call Graph from the SCG data. We have chosen these two models in addition to SCG, as both the Class Collaboration Network [31, 68, 76, 86, 100] and the Call Graph [3, 37, 46, 65, 98, 104] are models that are popular among researchers and have been used for software comprehension.

Our definition of the Class Collaboration Network is akin to the one presented by Wheeldon and Counsell [119]. Figure 8.1 illustrates the concept of CCN with three distinct class coupling types: inheritance, aggregation, and reference through parameter and return types. A CCN model is extracted from the SCG data by iteratively traversing relevant nodes, and creating a new graph comprising nodes of types *OBJECT*, *CLASS*, *TRAIT*, *INTERFACE*, and their associated edges: *INHERITANCE*, *AGGREGATION*, and *REFERENCE*.

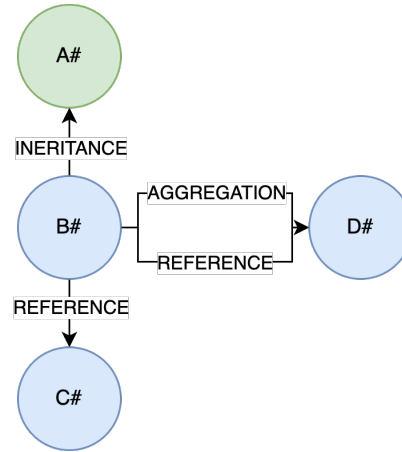
The Call Graph is obtained as a SCG subgraph with edges limited to type *CALL* and vertices of types *METHOD*, *CONSTRUCTOR*, *VALUE* and *VARIABLE*.

The remainder of this chapter is organized as follows. Section 8.2 presents the first case of software comprehension, where we show how simple analysis of project metrics, followed by more complex software mining, can lead to interesting insights. Section 8.3 provides an examination of the projects’ structural characteristics. Section 8.4 describes recovery of software architecture using graph partitioning algorithms. Section 8.5 continues with identification of critical code entities which typically serve as excellent starting

```

class D
class C
trait A
class B extends A {
  var d: D
  def bar(c: C): D = d
}

```



(a) Scala inheritance and aggregation example.

(b) Class Collaboration Network example visualization.

Figure 8.1: Class Collaboration Network example with *INHERITANCE* dependencies between *class B* and *class A*, *AGGREGATION* representing variable *D* and *REFERENCE* representing method param and return type coupling.

points for comprehending the source code. Section 8.6 demonstrates the capabilities of custom software mining analysis, specifically the identification of code similarities, made feasible by applying data analysis methods to the the rich SCG model.

8.2 Project overview

We start the software comprehension experiment with analysis of the simplest properties: the distribution of nodes (Fig. 8.2) and edges (Fig. 8.3) in the SCG graphs of the investigated projects.

Looking at the node distribution, we can immediately observe that all Java projects, except for *commons-io*, have a relatively high fraction of variables, reaching 22.5% for *spring-boot*, compared to only 6.3% in the case of *common-io*, and less than 4% for Scala projects. Since excessive use of variables can be harmful, we decide to use *software mining* techniques to investigate this further for the *spring-boot* project. First, we find top files in terms of the number of variables in the project. With the help of the *scg-cli* tool, we export the project SCG metadata into Jupyter notebook for closer inspection. The notebook provides a python environment with a helper API to read the SCG protobuf data into Pandas¹ data frames. Next, we use a standard split-apply-combine data analysis strategy, as shown in Listing 8.1. With this short program we can find 11807 local variables and discover the files with the highest number of them, e.g., *ConfigurationPropertyName.java*, *PaketoBuilderTests.java*, *JarIntegrationTests.java*, *OnBeanCondition.java* and *TomcatWebServerFactoryCustomizer.java*.

Listing 8.1: Dataframe-based analysis of the spring-boot SCG data

```

import scg

scg_files = scg.read_scg("data/spring-boot")

```

¹<https://pandas.pydata.org>

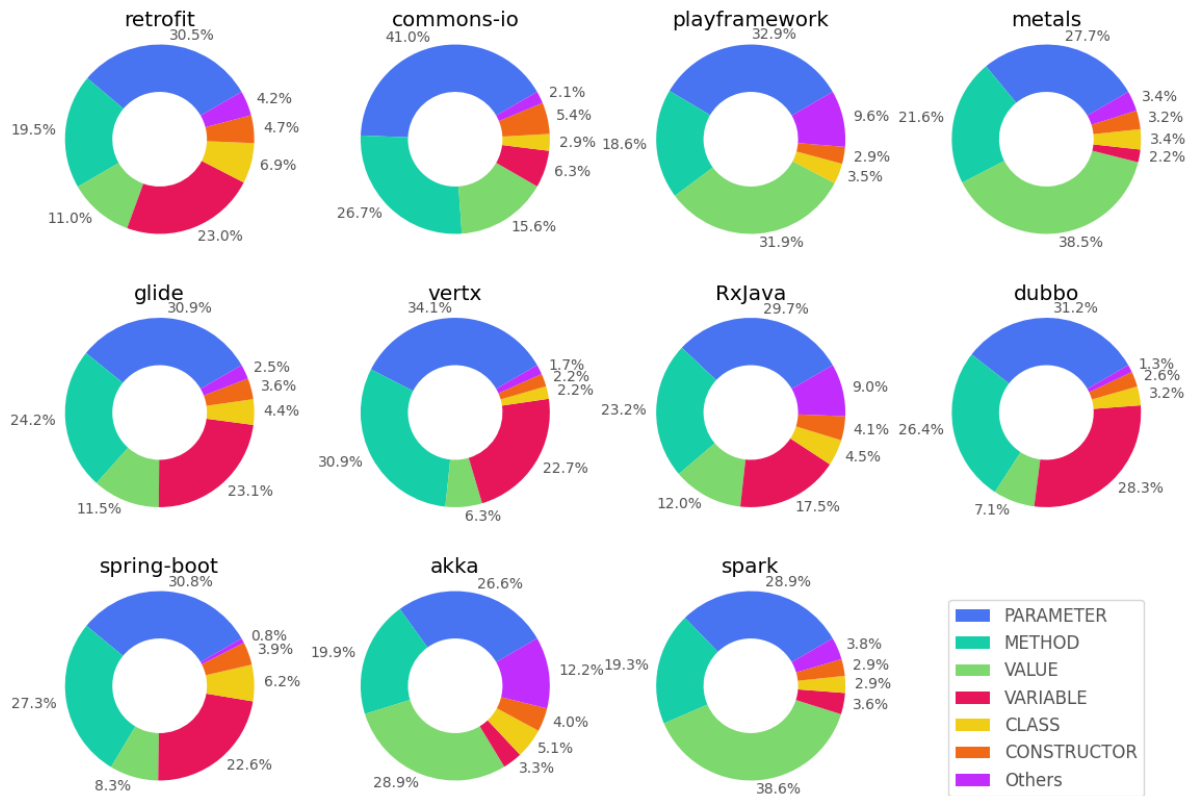


Figure 8.2: Top 6 SCG nodes distribution in analyzed projects.

```
df = scg.create_nodes_df(scg_files)
variables_df = df[df['kind']=="VARIABLE"]

variables_df.filter(items=['id', 'isLocal']) \
    .groupby('isLocal').count()
#           id
# isLocal
# false      2639
# true       11807

variables_df[variables_df['isLocal']==True] \
    .filter(items=['id', 'file']) \
    .groupby('file') \
    .agg(count=('id', 'count')) \
    .sort_values(by='count', ascending=False)
#           count
# file
# ConfigurationPropertyName.java      102
# PaketoBuilderTests.java             101
# JarIntegrationTests.java              86
# OnBeanCondition.java                 80
```

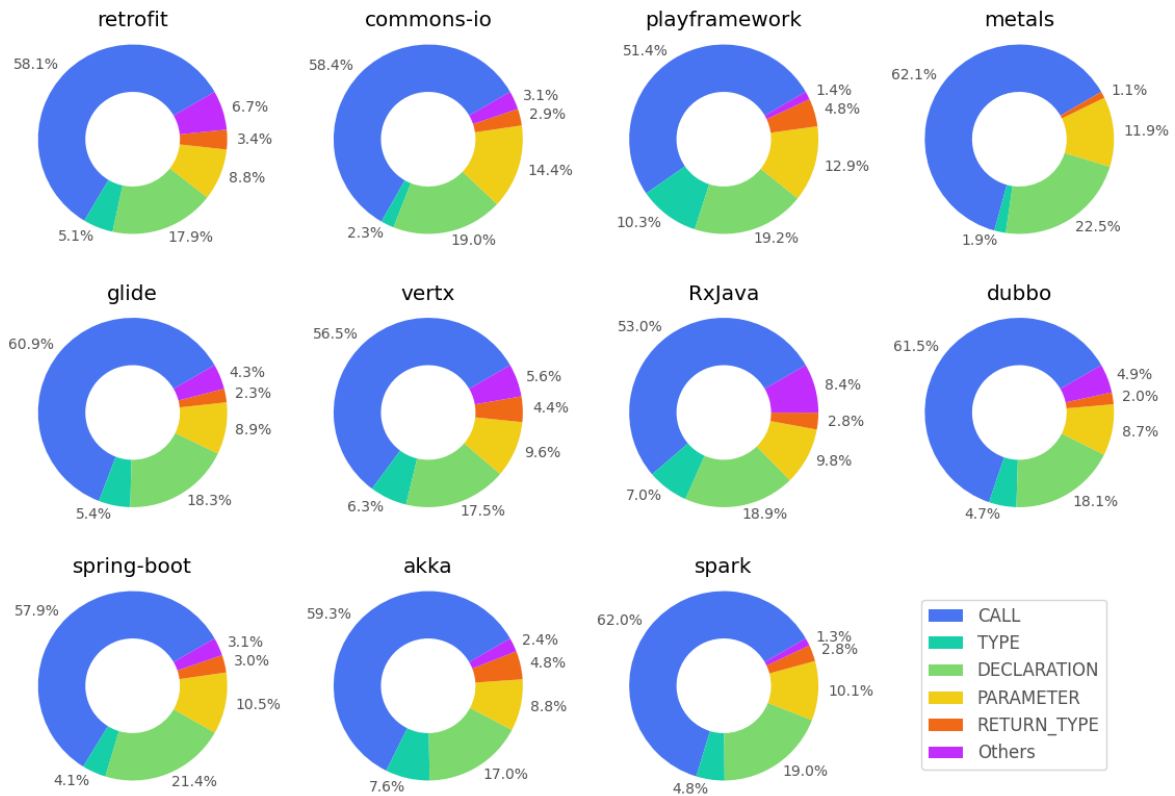


Figure 8.3: Top 5 SCG edges distribution in analyzed projects.

```
# TomcatWebServerFactoryCustomizer.java 72

variables_df[variables_df['isLocal']==False] \
    .filter(items=['id', 'file']) \
    .groupby('file') \
    .agg(count=('id', 'count')) \
    .sort_values(by='count', ascending=False)

#
# file
# ServerProperties.java 104
# KafkaProperties.java 76
# RabbitProperties.java 68
# FlywayProperties.java 60
# WebProperties.java 31
```

Looking at the source code of *ConfigurationPropertyName.java*, we can find many examples of mutable variables as presented in Listing 8.2 (e.g., `int start`) or immutable variables missing the `final` modifier (e.g., `int length`). It can be observed that many cases of the latter particularly contribute to the large number of variables. In comparison, the *common-io* project correctly uses the `final` keyword for each immutable local variable (see Listing 8.3), a good practice in the Java code, effectively transforming such

variables into immutable values. Consequently, we can conclude that the quality of the source code of the *spring-boot* project could be improved by eliminating the excessive number of local mutable variables.

Listing 8.2: Example of local variables' definitions in spring-boot *ConfigurationPropertyName.java* file

```
Elements parse(Function<CharSequence,
                CharSequence> valueProcessor) {
    int length = this.source.length();
    int openBracketCount = 0;
    int start = 0;
    ElementType type = ElementType.EMPTY;
    ...
}
```

Listing 8.3: Example of final local variables definition in *commons-io* project

```
public static void copy(
    final String input,
    final OutputStream output,
    final String encoding)
    throws IOException {
    final StringReader in =
        new StringReader(input);
    final OutputStreamWriter out =
        new OutputStreamWriter(output, encoding);
    copy(in, out);
    out.flush();
}
```

A different case of using variables that we have found are non-local variables present in *spring-boot* property files, such as *ServerProperties.java*, *KafkaProperties.java*, *RabbitProperties.java*, *FlywayProperties.java*. These files contain different configuration settings, such as the defined **port** field presented in Listing 8.4, which are accessed through global accessors. While this has been a standard Java approach for classes representing some internal state, currently the recommended practice is to introduce specialized constructors and add **final** modifiers for global private fields to eliminate accidental mutability. Performing such refactoring is another example of recommendation that we can formulate in order to increase the quality of the project source code.

Listing 8.4: Example of global variable definition in spring-boot *ServerProperties.java* file with accessors methods

```
private Integer port;

public Integer getPort() {
    return this.port;
}

public void setPort(Integer port) {
    this.port = port;
}
```

8.3 Project structure comprehension

In this example, we demonstrate how project structure discovery can be performed by looking at various graph metrics, described in Table 8.1. For each project, we computed all metrics for the full SCG, the Class Collaboration Network and the Call Graph. The results are presented in Table 8.2.

The metrics show that for Java projects, there are on average more lines of code per graph node (code entity) than in Scala projects. This is expected as the Scala language is much more concise than Java. Nevertheless, the difference is quite significant with around 1.7 times as many code lines per node for Java projects. These metrics suggest that the Scala language can greatly contribute to lower code verbosity.

All projects have low *density*, although the SCG graph for the *retrofit* project is around 55 times denser than one for the *spark* project. This metric is much higher also for CCN and CG models, possibly indicating a higher overall coupling level.

The incoming and outgoing degree distribution of SCG nodes, as presented in Figure 8.4, follows a power-law distribution for each project, indicating that the network exhibits characteristics of a scale-free network [49, 79]. This is further supported by the low average node degree for each network, combined with a high standard deviation (σ) and a high index of dispersion (*IoD*) for both incoming and outgoing degree distributions. A scale-free network indicates the likely existence of highly connected code entities (hubs), which may represent core functionalities. Changes, removals, or vulnerabilities in relation to random low-level nodes are likely to have minimal effects on the software. Conversely, altering hubs can significantly impact the majority of the software in terms of structure, correctness and program execution performance. Upon examining all eleven analyzed projects, it becomes evident that the characteristics of node distribution are not correlated with either the project size or the project's programming language. Additionally, although the average degree for each project is similar (approximately 3), the standard deviation and index of dispersion can vary significantly. A particularly high index of dispersion may signal the presence of outlier nodes with exceptionally high degree levels.

For example, the *Retrofit* project has an exceptionally high index of dispersion (third highest) for the outgoing node degree, observed both in SCG and CG networks, with values of 57.4 and 52.1, respectively. At the same time, this parameter is very small for the CCN network (the lowest), which is different to what we can observe for other projects. When examining the node degree distribution of the *Retrofit* project, as presented in Figure 8.5, we observe an outlier node which corresponds to *Builder#parseParameterAnnotation*, a 455 lines-of-code method that has an extremely high outgoing node degree of 674. Degree and size of this method alone seems to indicate high method complexity and bad programming practices. *Retrofit* is an HTTP client that builds a client service based on a Java interface augmented by HTTP-related annotations. The *Builder#parseParameterAnnotation* method matches the annotations and builds the appropriate *ParameterHandler*. We recommend refactoring this method into smaller methods, with each method responsible for handling a particular annotation type and transforming it into a *ParameterHandler*. This will improve code readability, reduce the method's complexity, and enhance the source code quality. It is worth noting that the outlier observed in the *Builder#parseParameterAnnotation* method could not be detected using the CCN model, as CCN operates at the abstract class level and does not consider method bodies or, in particular, local variables.

Property	Explanation
Density (D)	$ D = \frac{ E }{ V (V - 1)}$ Density of the graph – measure of how closely connected the graph is. High SCG density can indicate higher code dependency complexity. $D \in [0, 1]$
Average Node Degree (A_D)	$A_D = \frac{ E }{ V }$ High average degree of nodes, especially combined with high density, suggests high project coupling; on the contrary, high node average degree with low density might imply a significant number of hubs.
Standard Deviation of incoming/outgoing node degree distribution (σ_{ID}/σ_{OD})	$\sigma = \sqrt{\frac{1}{ V } \sum_{i=1}^{ V } (d_i - A_D)^2}$ Where: • V is number of nodes • d_i is an incoming or outgoing degree of a node i A high standard deviation in the node degree distribution indicates significant variability in node degrees. This may suggest that certain entities exhibit considerably higher incoming or outgoing degrees than the average. Consequently, a high standard deviation might indicate greater code complexity and coupling.
Index of Dispersion of incoming/outgoing node degree distribution (IoD_{ID}/IoD_{OD})	$IoD = \frac{\sigma^2}{A_D}$ Where: • σ^2 is the variance of the node degrees (incoming or outgoing) Related to Standard Deviation. If the IoD is greater than 1, it indicates that the node degrees are more variable (dispersed) than what one would expect from a Poisson distribution. A high IoD suggests that the network may possess a scale-free structure, characterized by a few highly connected nodes (hubs) and numerous nodes with lower degrees.
Average Clustering Coefficient (ACC)	$ACC = \frac{1}{ V } \sum_{i=1}^{ V } \frac{2T_i}{d_i(d_i - 1)}$ Where: • T_i represents the number of triangles (closed triplets) that node i is part of. • d_i represents the degree of node i The average clustering coefficient provides an average measure of how tightly interconnected nodes are in local neighborhoods across the entire network. $ACC \in [0, 1]$
Global Clustering Coefficient (GCC)	$GCC = 3 \times \frac{\text{number_of_triangles}}{\text{number_of_triplets}}$ High value indicates that graph tends to create interconnected clusters, low value indicates the network being sparse or fragmented. For SCG high GCC value can indicate well-modularized software with high module cohesion and low coupling. $GCC \in [0, 1]$
Degree Assortativity Coefficient (DAC)	$DAC = \frac{\sum_{i,j} (A_{ij} - d_i k_j / 2m) d_i d_j}{\sum_{i,j} (A_{i,j} d_i^2 - \frac{d_i d_j}{2m} d_i d_j)}$ where [81]: • A is the adjacency matrix • d_i is the degree of the node i • m is the total number of edges Positive value indicates that nodes with the same degree tends to be connected, negative value suggest mostly connections between nodes of different degree, value close to zero suggest random network. For SCG low DAC can suggest additional tendency of the graph towards clustering and establishing hubs (connection between nodes with different degree). $DAC \in [-1, 1]$

Table 8.1: Metrics calculated for the SCG, CCN and CG software networks.

Name	Version	#LOC	V	$\frac{\#LOC}{ V }$	E	D	A_D	σ_{ID}	IoD_{ID}	σ_{OD}	IoD_{OD}	ACC	GCC	DAC
Semantic Code Graph														
retrofit	2.9.0	13676	3035	4.51	10137	0.00110	3.3	6.2	11.7	13.8	57.4	0.16	0.03	-0.05
commons-io	2.12.0	46526	7418	6.27	21257	0.00039	2.9	3.3	3.7	7.2	17.8	0.13	0.07	-0.23
playframework	2.8.19	57592	23058	2.50	67080	0.00013	2.9	9.1	28.6	10.5	38.0	0.10	0.03	-0.08
metals	0.10.3	58172	19511	2.98	53347	0.00014	2.7	3.2	3.8	10.5	40.7	0.08	0.03	-0.14
glide	4.5.11	65397	15459	4.23	53211	0.00022	3.4	5.3	8.3	8.3	20.1	0.15	0.08	-0.27
vert.x	4.4.4	102189	24498	4.17	86664	0.00014	3.5	12.5	44.1	10.8	33.2	0.13	0.03	0.17
RxJava	3.1.6	188189	37729	4.99	118788	0.00008	3.1	11.9	45.2	7.7	19.0	0.12	0.02	-0.06
dubbo	3.2.4	247943	57601	4.30	203693	0.00006	3.5	11.2	35.5	9.3	24.5	0.14	0.04	-0.04
spring-boot	2.7.5	330424	64190	5.15	186816	0.00005	2.9	4.0	5.4	6.0	12.3	0.16	0.09	-0.12
akka	2.7.0	337559	122355	2.76	436691	0.00003	3.6	17.8	88.9	20.3	115.9	0.12	0.01	-0.05
akka (without tests)	2.7.0	234445	90195	2.60	295957	0.00004	3.3	13.4	54.5	11.3	38.9	0.12	0.02	-0.08
spark	3.3.0	527149	164772	3.20	561314	0.00002	3.4	16.9	84.0	16.7	81.9	0.09	0.01	-0.03
Class Collaboration Network														
retrofit	2.9.0	13676	183	74.73	473	0.01420	2.6	12.2	57.5	2.0	1.5	0.23	0.08	-0.61
commons-io	2.12.0	46526	174	267.39	438	0.01455	2.5	8.8	30.9	4.9	9.7	0.09	0.09	-0.22
playframework	2.8.19	57592	1208	47.68	4935	0.00338	4.1	29.3	209.9	19.2	89.9	0.09	0.02	0.10
metals	0.10.3	58172	680	85.55	1427	0.00309	2.1	4.6	10.1	3.2	4.8	0.05	0.10	-0.61
glide	4.5.11	65397	707	92.50	2597	0.00520	3.7	11.6	36.7	5.5	8.4	0.16	0.15	-0.42
vert.x	4.4.4	102189	600	170.32	4960	0.01380	8.3	39.3	186.6	13.4	21.8	0.18	0.46	-0.30
RxJava	3.1.6	188189	1564	120.33	6787	0.00278	4.3	36.1	299.7	11.0	28.1	0.16	0.84	-0.40
dubbo	3.2.4	247943	2030	122.14	8641	0.00210	4.3	33.1	256.6	5.8	7.8	0.15	0.03	-0.17
spring-boot	2.7.5	330424	3199	103.29	7096	0.00069	2.2	6.1	16.8	3.5	5.5	0.10	0.09	-0.14
akka	2.7.0	337559	8107	41.64	27250	0.00041	3.4	30.4	274.4	12.9	49.9	0.06	0.19	-0.09
akka (without tests)	2.7.0	234445	5477	42.81	21254	0.00071	3.9	31.4	253.7	15.5	62.2	0.07	0.25	-0.05
spark	3.3.0	527149	6461	81.59	31742	0.00076	4.9	54.1	594.8	13.6	37.6	0.12	0.06	-0.19
Call Graph														
retrofit	2.9.0	13676	1677	8.16	4151	0.00148	2.5	4.1	6.9	11.4	52.1	0.13	0.03	-0.06
commons-io	2.12.0	46526	3604	12.91	7232	0.00056	2.0	3.2	5.0	4.9	12.0	0.08	0.06	-0.22
playframework	2.8.19	57592	9469	6.08	16750	0.00019	1.8	5.9	19.9	6.5	24.0	0.01	0.01	-0.05
metals	0.10.3	58172	10900	5.34	20638	0.00017	1.9	2.9	4.4	7.9	33.4	0.02	0.01	-0.11
glide	4.5.11	65397	8863	7.38	24807	0.00032	2.8	5.0	8.9	7.2	18.4	0.12	0.06	-0.22
vert.x	4.4.4	102189	13431	7.61	33828	0.00019	2.5	5.4	11.6	6.4	16.5	0.10	0.05	-0.22
RxJava	3.1.6	188189	19348	9.73	43800	0.00012	2.3	5.3	12.5	5.9	15.2	0.10	0.04	-0.14
dubbo	3.2.4	247943	33038	7.50	91100	0.00008	2.8	7.5	20.2	7.4	19.7	0.12	0.04	-0.14
spring-boot	2.7.5	330424	36803	8.98	73727	0.00005	2.0	3.8	7.3	4.4	9.8	0.10	0.07	0.02
akka	2.7.0	337559	47621	7.09	107184	0.00005	2.3	12.5	69.8	7.0	21.8	0.01	0.01	-0.05
akka (without tests)	2.7.0	234445	37916	6.18	87809	0.00006	2.3	11.9	61.5	7.3	23.3	0.01	0.01	-0.05
spark	3.3.0	527149	90285	5.84	208191	0.00003	2.3	10.1	44.1	9.7	40.9	0.01	0.01	-0.08

Table 8.2: Project characteristics extracted from the Semantic Code Graph, Class Collaboration Network and the Call Graph (D = Density, A_{OD} = Average out-degree, A_{ID} = Average in-degree, σ = Standard Degree Distribution Deviation, IoD = Degree Distribution Index of Dispersion, ACC = Average Clustering Coefficient, GCC = Global Clustering Coefficient, DAC = Degree Assortativity Coefficient)

For the SCG graph, the highest Index of Dispersion can be observed for *akka* project, with $IoD_{ID} = 88.9$ and $IoD_{OD} = 115.9$ which is much higher than in the case of all other projects. A detailed distribution of node degrees, presented in Fig. 8.4, shows that a significant number of nodes have degrees above 100 and there are several nodes with outgoing and incoming degrees above 1000. Notable nodes include the method responsible for sending messages in the *akka* toolkit, *ActorRef#!* with an incoming degree of 2747, the *ActorRef* class representing actor references with a degree of 1898, and the *GraphStageLogic#pull* method responsible for pulling new messages in *akka* streams. Similar observations can be made for the CCN and CG networks. However, it is worth noting that CCN cannot identify critical methods, while CG falls short in identifying key class components. In this regard, the SCG information model provides the most comprehensive insight into important project outliers. Regarding IoD_{OD} , upon closer examination at node distribution

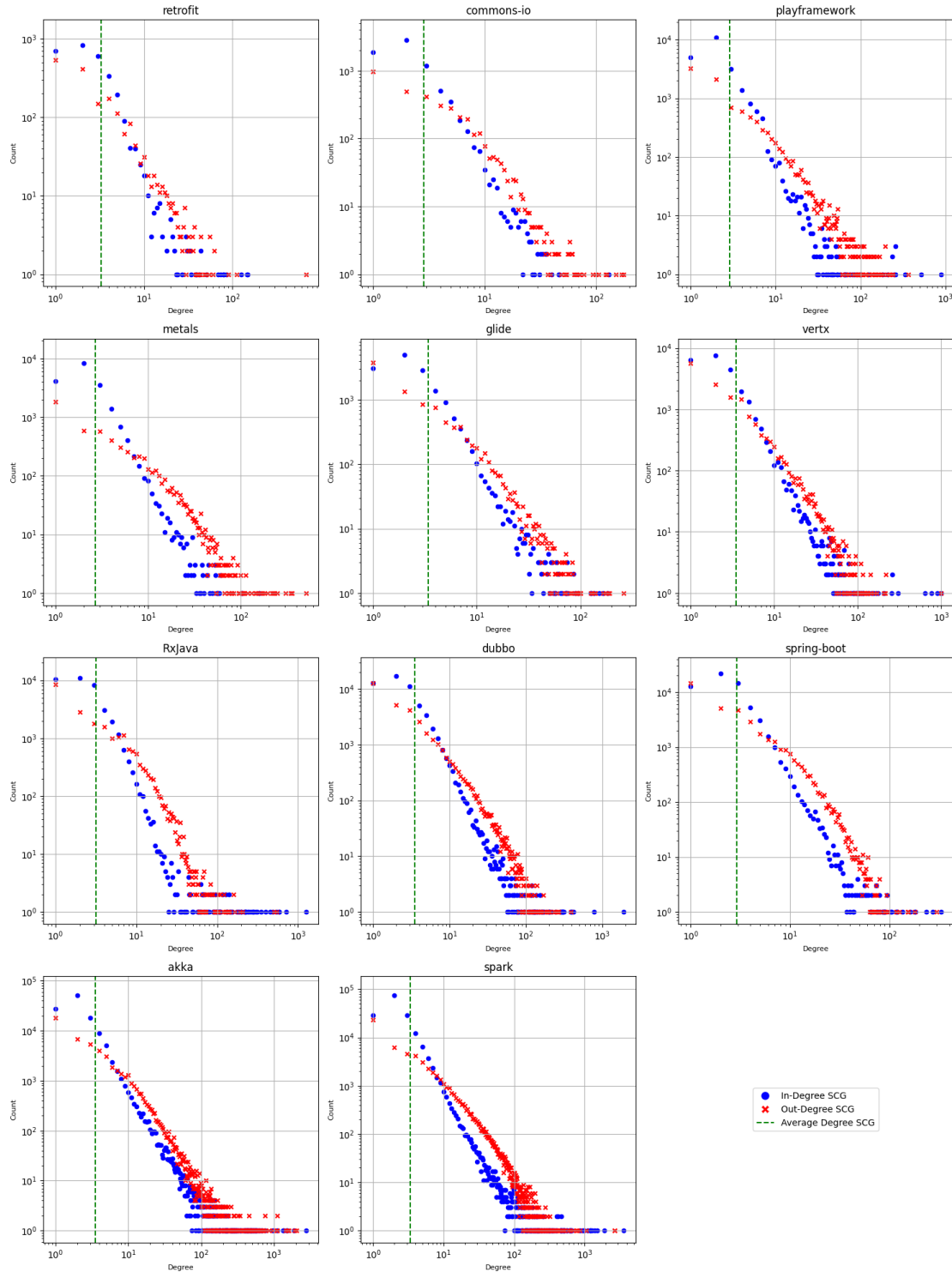


Figure 8.4: Nodes' incoming and outgoing degree distribution for SCG model extracted for each analyzed project.

presented at Fig. 8.6, it appears to be primarily influenced by large test classes. Test cases implemented with the *scalatest* library reside within the test class body and contribute to the high outgoing node degree. Example include class *ByteStringSpec* with 2031 outgoing connections and class *ORMapSpec* with 1939 outgoing connections. We have also added project characteristics computed for the akka project without tests, which

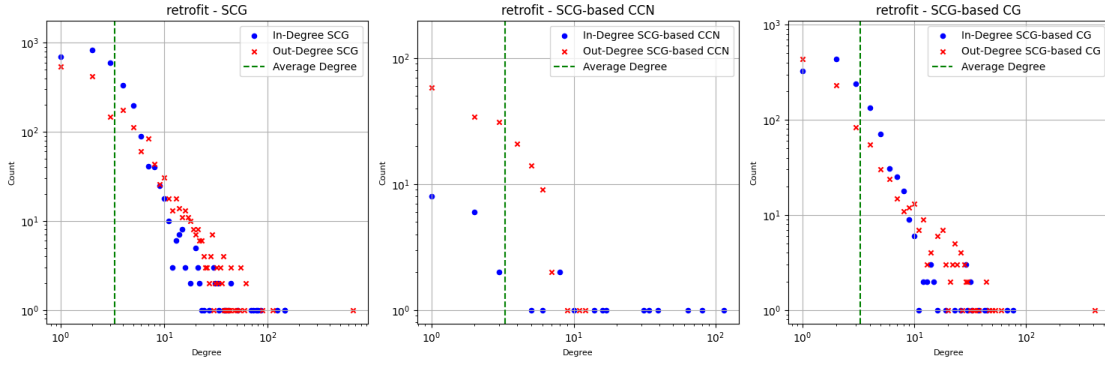


Figure 8.5: The incoming and outgoing degree distribution of nodes in the SCG, CCN, and CG models was extracted for the *retrofit* project, with a focus on identifying outlier nodes with high outgoing degrees in the SCG and CG networks.

confirms big influence of test files on project structure and node distribution – the value of IoD_{OD} drops from 115.9 to 38.9. Interestingly, for CCN and CG models the network characteristics do not change much for akka whether with or without tests. It may be an interesting avenue for future research to investigate the impact of tests on the project structure.

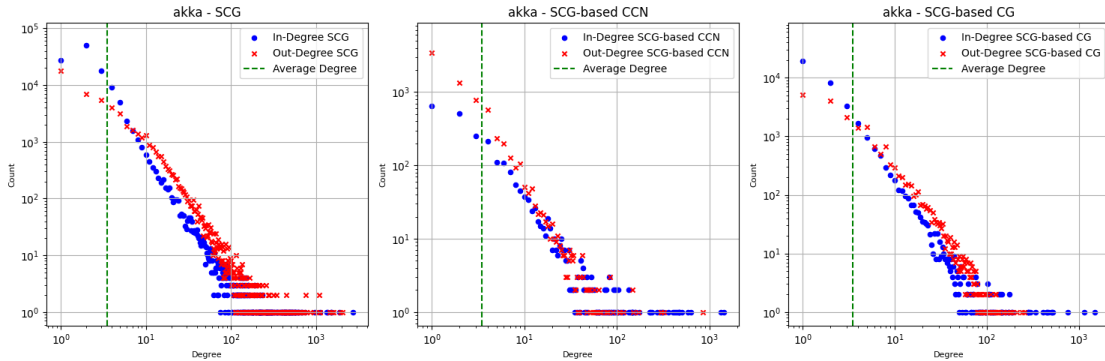
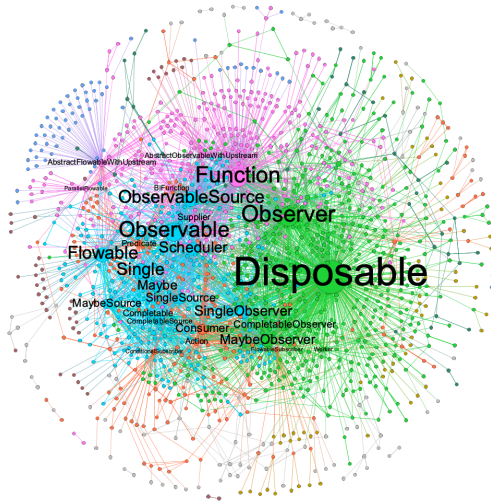


Figure 8.6: The incoming and outgoing degree distribution of nodes in the SCG, CCN, and CG graphs extracted for the *akka* project. Nodes with high incoming degrees, such as the *ActorRef* class and the method *ActorRef#!*, serve as the primary building blocks for the *akka* project.

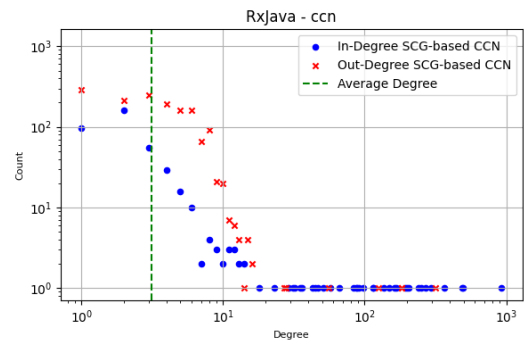
The low value of the Global Clustering Coefficient (GCC) indicates that the software forms a sparse network and does not manifest clustering tendency. A high GCC value would denote a high project clustering tendency, indicating that the network is well connected on the global level. On the other hand, the Average Clustering Coefficient (ACC) measures local clustering around individual nodes and places more weight on low-degree nodes. A high value of ACC would mean that the network is tightly connected, which violates good software practices and complicates software maintainability in the long run [15]. All eleven analyzed projects have relatively low GCC and ACC values for both SCG and CG representations. Consequently, it should be expected that refactoring tasks related to software modularization should not be hindered by software structure. For CCN, the GCC values are higher, which is expected, as during CCN creation, the algorithm effectively applies path contraction to mark direct class dependencies and filter out entities other

than classes. Class nodes contribute the most to the global structure and, hence, to clustering tendency. This naturally tightens the dependencies between entities and increases the clustering likelihood.

The *RxJava* project presents an interesting case, where we observe the largest spike of the GCC value (CCN model), reaching 0.84. High GCC and low ACC are typical characteristics of a small-world network, where most of the nodes are part of tightly-connected clusters, but there are a few hubs that bridge these clusters and shorten the path between otherwise distant nodes. A very high index of dispersion for incoming node degree equal to 299.7 might indicate that these hubs are nodes with particularly high incoming degree. Indeed, in Figure 8.7, we can visually observe the project's tendency for global clustering with a few hub nodes, such as *Disposable*, *Observer*, *Function*, and *ObservableSource*, being the top in-degree nodes. We can conclude that the CCN network, although extracted directly from the SCG model, provides an interesting perspective, showing the otherwise not visible clustering tendency of the *RxJava* project at the class-level granularity.



(a) Visualization of the CCN for the *RxJava* project reveals a clear tendency toward global clustering, confirmed by a high GCC value of 0.84. It also visually confirms the small-world network property. Low degree nodes on the outside contribute to an overall low ACC value.



(b) Degree Distribution for *RxJava* project with visible high degree outliers.

Figure 8.7: CCN visualization and degree distribution for the *RxJava* project confirming high tendency for global clustering.

These and similar metrics, apart from software comprehension activities, can also be used to assess the effort needed for certain refactoring activities or to measure the quality of the project structure throughout the evolution of the software [15]. For example, increasing density and DAC with decreasing GCC can indicate project quality degradation.

8.4 Software architecture recovery through software module clustering

Software Module Clustering [99] is the process of dividing software into smaller groups that share common features and exhibit cohesion while remaining loosely coupled [78, 91]. Software Module Clustering is often considered a refactoring activity aimed at identifying smaller modules that are easier to develop and evolve. However, it can also directly contribute to software comprehension through architecture recovery [16, 35, 102, 122], the discovery of similar structures [110], the measurement of software evolution [52], and the detection of problems related to design quality [121].

In recent years, another field has emerged where Software Module Clustering can be effectively applied: migrating monolithic applications to separate, cohesive smaller services, often referred to as microservices [30, 88]. The first and indispensable step in transitioning from a monolithic architecture to a microservice architecture is to identify a well-defined, cohesive part of the monolith and extract it into a separate service [36]. With a well-modularized monolith, a small module naturally becomes a candidate for migration into a separate service. However, monolithic applications often degrade in quality, and the module structure and also intermodule dependencies become complex, coupled and difficult to understand [62], posing challenges in identifying the initial part of the software to be extracted [53]. This identification problem can be addressed through a software split into two, often imbalanced in size, parts, with additional modularization applied later². This approach allows also for gradual refactoring in an incrementally manageable manner. If the project split is hard or even impossible, which should be reflected in poor partition quality, Software Module Clustering can guide and supervise refactoring efforts towards better modularization before the actual migration. Similarly, when aiming to discover the software architecture, our goal is to identify the main parts of the software and establish clear boundaries between different system components. With the software source code represented as a detailed graph model, such as SCG, popular graph partitioning algorithms can be applied with high precision at a granular level of the code base entities. For software networks, the intuitive objective is to minimize the cut size of the partitions, measured by the number of edges crossing different partitions. This optimization ensures that the partitions remain loosely coupled, facilitating their extraction into separate modules with minimal inter-module interactions. Optimizing the cut size is a typical approach in *graph partitioning* [26]. However, while graph partitioning usually also aims to create partitions similar in size, in the domain of software architecture recovery, having well-balanced partitions is not necessarily desirable — it is quite common for some cohesive parts of the system to be much larger than others. Nevertheless, by partitioning the graph into a small number of parts and permitting significant partition imbalance, we should be able to effectively uncover the true software architecture and obtain practical modularization insights.

For the largest analyzed projects, the size of the SCG graph reaches hundred thousands of nodes and edges (our largest analyzed project, Apache Spark, has more than 164k nodes and 561k edges). Fortunately, while graph partitioning is an NP-hard problem, there exist very fast heuristics that produce high-quality solutions and tools that implement them. In our experiments, we use two such tools – *gpmetis* [57] and

²<https://martinfowler.com/articles/break-monolith-into-microservices.html>

patoh [27, 28]. Both of them implement the heuristic based on multilevel partitioning algorithm involving three main phases:

1. *Coarsening Phase* – during this phase the size of the graph is gradually decreased.
2. *Initial Partitioning Phase* – an initial partitioning is computed on a smaller graph obtained from the Coarsening Phase.
3. *Uncoarsening Phase* – the initial partitioning is successively refined and is projected to the larger graph to eventually match the provided graph and all the nodes are assigned to some partition.

The *gpmets* program takes a graph as input and computes the requested number of partitions. It has several important arguments that can improve the results and can be adjusted specifically for software network partitioning. To apply *gpmets* to our software partitioning use case, the following options can be defined:

- *-ptype* – specifies the type of the partitioning algorithm, with possible options being recursive bisectioning and direct k-way partitioning. Since software networks are irregular, we use the default k-way partitioning.
- *-objtype* – specifies the objective that the partitioning algorithm will optimize, either the edge cut size or the communication volume. For partitioning the software structure, optimizing the edge cut size is the preferable option as it entails fewer direct dependencies on other modules.
- *-contig* – specifies whether partitioning should attempt to preserve contiguous partitions. Contiguity is a desired trait for software modularization.
- *-ufactor* – specifies the maximum allowed load imbalance among the partitions. In our case, any imbalance is allowed, but it can be adjusted according to the requirements.
- *-ncuts* – specifies how many times the algorithm will attempt to compute the partitioning. The final result is the partition that achieves the best edge cut size.

The *patoh* program operates on hypergraphs [23]. Our SCG, CCN, and CG models can be converted into hypergraphs by applying two rules: (1) treating the base graph as a hypergraph where each pair of nodes is used as a separate hypergraph edge; (2) for each node in the graph, creating a single hypergraph edge for all outgoing SCG edges of a particular type connecting this node with other nodes. This approach preserves the original graph characteristics while also emphasizing that all outgoing edges of the same type should be assigned to the same partition.

For the *patoh* program, important command-line arguments include:

- *IB* – controls the initial and final partition imbalance. It allows for flexibility in partition size or the enforcement of required balance.
- *PA* – Partition Algorithm – offers a choice of 13 different variants of initial partitioning algorithms.

We measure the quality of the obtained partitioning with the eight different metrics:

- Modularity – measures the ratio of edges between nodes inside a partition to edges crossing the partition boundary.
- Edge cut size – quantifies the number of edges that pass through the cut between different graph partitions.
- File/Package Weighted/Standard Average Accuracy – calculates the average percentage of nodes within a given file or package unit that belong to the same partition. In a well-structured software, all nodes in a particular file or package should be assigned by the algorithm to the same partition. Any outstanding nodes could indicate a need to reorganize the source code in order to improve the quality of the project structure. Weighted Accuracy additionally considers the size (in terms of the total number of nodes) of a particular file or package.
- Coefficient of variation – measures the similarity of partition sizes. A value close to zero indicates that the number of nodes in all partitions is roughly equal, while a value close to 1 suggests highly unbalanced partitions in terms of the size.
- Node distribution – expresses the spread of nodes across partitions as a fraction of the total node count.

Modularity, edge cut size, and the coefficient of variation are recognized metrics for assessing partition quality. Additionally, we propose four new domain-specific quality metrics that aim to evaluate how well the computed split aligns with the current structure of files and packages. We assume that the software is well-structured with respect to file and package units. Furthermore, if we were to modularize the software, we would prefer to do it on the basis of entire files or packages, minimizing the need for moving code entities and refactorings. Ideally, all nodes (code entities) within a file or package should belong to the same partition.

As demonstrated in Section 7.4.4, partitioning using the *gpmetis* and *patoh* programs can be directly performed on the SCG, CCN and CG models through the *scg-cli* tool, allowing for export in *graphml*, *.GDF*, and *CSV* formats. The *scg-cli* exposes all the aforementioned *gpmetis* and *patoh* parameters, making it easy for end-users to experiment with various partition numbers and sizes. Additionally, the *scg-cli* provides all the partition quality metrics described above.

For our empirical investigation, we focus on the *metals* project and, using partitioning, we try to discover main cohesive parts of this project. We allow for imbalanced partitions as we aim purely for minimal edge cut size, as it helps to discover the actual software architecture as explained above. In extreme cases the cut size might be equal to 0. This is due to the fact that the graph network may have multiple distinct connected components, forming natural partitions. Graphs might be disconnected for various reasons, such as the presence of isolated classes, stand-alone scripts, or – in the case of the CCN network – lack of structural dependencies between entities (which may be connected only by call relationships, not reflected in the CCN).

Using an *scg-cli* tool command, similar to one presented in Listing 8.5, we can determine the split of the *metals* project into a given number of partitions (8 in this case), for one of three graph representations of this software – the full SCG, the CCN or the CG network.

Listing 8.5: *scg-cli* command to assess *metals* partitioning up to 8 partitions with allowed significant imbalance for both *patoh* and *gpmetis* algorithms

```
scg-cli partition -o graphml -g SCG -ot cut -uf 999 -ib 0.9 data/metals-0.10.3.zip 8
```

The quality metrics for the obtained partitions of each type of network are presented in Table 8.3. Based on the data presented, we can conclude that both *patoh* and *gpmetis* behave similarly and produce partitions of similar quality. However, there are cases where one tool outperforms the other significantly. For instance, in the SCG graph split into three partitions, *patoh* achieves an edge cut of 418, while *gpmetis* achieves 638. Interestingly, this significant difference in modularity does not necessarily entail better domain-specific metrics. On the contrary, usually partitioning the hypergraph representation produces notably worse results in terms of the domain quality despite the much lower edge cut size. For the mentioned SCG split into three partitions, the PWAA metric decreases from 86% to 82%. This simple example shows that each partitioning result has to be carefully considered taking into account all specific requirements to assess which partitioning result is preferable.

The general degradation in the PWAA quality metrics as the number of partitions increases suggests that aligning partitions with the current package structure is challenging. Conversely, file-based alignment remains consistently high (above 90% for the SCG) and degrades much more slowly with the increasing number of partitions. This indicates that, in case of refactoring, reorganizing the project based on files should be easier to execute, while packages might need some reorganization. Additionally, it is worth noting that the larger the number of partitions, the larger the expected edge cut size becomes. To help with the modularization effort, *scg-cli* can suggest which package should be placed in which partition. Table 8.4 presents such information for the *metals* project split to three partitions using *patoh*. Such a summary, visualized on the software graph presented in Fig. 8.8, provides insight into software architecture and enables further in-depth analysis.

Method	N	Modularity	Cut Size	FWAA %	FAA %	PWAA %	PAA %	CV	Distribution %
Semantic Code Graph									
gpmetis	2	5960.2	9	99	99	98	96	0.865	[6,93]
patoh	2	6705.4	8	99	99	98	97	0.864	[93,6]
gpmetis	3	83.1	638	96	98	86	93	0.706	[16,16,66]
patoh	3	127.4	418	97	98	82	91	0.658	[9,62,28]
gpmetis	4	47.8	1099	95	97	80	87	0.661	[6,13,29,49]
patoh	4	58.2	906	95	97	75	83	0.551	[21,47,21,9]
gpmetis	5	36.3	1439	95	97	78	90	0.782	[6,10,4,39,38]
patoh	5	38.1	1373	93	96	68	82	0.498	[15,10,12,23,37]
gpmetis	6	33.1	1572	94	96	74	86	0.586	[6,10,9,13,26,33]
patoh	6	32.2	1616	92	96	68	83	0.471	[10,31,19,6,16,14]
gpmetis	7	27.7	1871	93	96	69	84	0.561	[2,8,13,8,28,19,19]
patoh	7	26.4	1960	92	96	59	78	0.429	[9,7,12,10,16,16,27]
gpmetis	8	23.8	2161	91	95	64	80	0.473	[10,13,7,13,6,6,17,24]
patoh	8	23.7	2168	91	95	54	78	0.410	[14,9,14,23,9,6,13,8]
Class Collaboration Network									
gpmetis	2	Infinity	0	98	99	90	95	0.391	[30,69]
patoh	2	Infinity	0	98	99	90	95	0.391	[69,30]
gpmetis	3	Infinity	0	98	99	89	94	0.777	[19,11,69]
patoh	3	52.0	27	98	99	80	92	0.329	[30,47,21]
gpmetis	4	42.4	33	98	99	77	89	0.580	[14,15,20,49]
patoh	4	38.8	36	97	98	76	87	0.525	[21,47,15,15]
gpmetis	5	20.7	66	97	98	73	87	0.647	[12,12,4,39,30]
patoh	5	16.9	80	96	98	72	86	0.394	[15,15,13,34,21]
gpmetis	6	16.2	83	97	98	71	85	0.512	[12,8,16,8,20,32]
patoh	6	14.4	93	95	97	66	80	0.375	[10,29,18,14,13,13]
gpmetis	7	10.3	127	95	98	68	81	0.418	[12,12,12,28,8,11,13]
patoh	7	11.3	116	95	97	65	81	0.383	[10,11,9,26,16,13,12]
gpmetis	8	9.8	133	95	97	65	80	0.385	[24,9,10,12,10,11,8,12]
patoh	8	9.2	140	95	97	63	80	0.315	[22,13,11,10,11,9,11,9]
Call Graph									
gpmetis	2	Infinity	0	96	95	93	90	0.888	[5,94]
patoh	2	Infinity	0	96	95	93	91	0.881	[94,5]
gpmetis	3	110.6	187	92	92	81	85	0.714	[12,21,66]
patoh	3	139.0	149	92	93	80	83	0.643	[10,27,61]
gpmetis	4	40.3	505	90	92	72	79	0.584	[19,17,12,49]
patoh	4	53.6	382	91	91	70	78	0.531	[21,47,20,11]
gpmetis	5	41.0	497	88	90	69	77	0.744	[10,11,3,35,39]
patoh	5	35.0	580	87	90	68	77	0.471	[13,12,12,23,36]
gpmetis	6	29.7	680	87	90	65	78	0.687	[1,14,33,5,15,29]
patoh	6	28.2	714	84	88	56	71	0.445	[10,19,31,10,16,11]
gpmetis	7	25.9	775	85	89	55	71	0.646	[5,6,12,11,7,28,28]
patoh	7	25.0	803	85	87	54	71	0.424	[9,12,7,16,10,16,26]
gpmetis	8	22.7	882	84	88	57	75	0.599	[13,6,6,11,6,6,24,24]
patoh	8	23.2	864	81	86	52	68	0.385	[14,23,10,13,11,7,11,7]

Table 8.3: Quality metrics for different partitionings of the metals project using SCG, CCN and CG networks with *patoh* and *gpmetis* software, invoked through the *scg-cli* tool. (N – the number of partitions, Cut Size – the number of edges crossing partitions, FWA – File Weighted Average Accuracy, FAA – File Standard Average Accuracy, PWAA – Package Weighted Average Accuracy, PAA – Package Standard Accuracy, CV – Coefficient of variation)

Method	Part.	Accuracy %	Package	Distribution
patoh	0	100	example	[119,0,0]
patoh	0	100	example.nested	[13,0,0]
patoh	0	100	example.nested.x	[2,0,0]
patoh	0	100	example.nested2.y	[2,0,0]
patoh	0	100	example.type	[2,0,0]
patoh	0	100	metals	[7,0,0]
patoh	0	60	scala.meta.internal.async	[6,4,0]
patoh	0	100	scala.meta.internal.docstrings	[409,0,0]
patoh	0	100	scala.meta.internal.sbtmetals	[4,0,0]
patoh	1	53	docs	[66,75,0]
patoh	1	100	scala.meta.internal.ansi	[0,31,0]
patoh	1	98	scala.meta.internal.bsp	[2,116,0]
patoh	1	100	scala.meta.internal.builds	[0,439,0]
patoh	1	100	scala.meta.internal.decorations	[0,132,0]
patoh	1	84	scala.meta.internal.implementation	[0,278,50]
patoh	1	83	scala.meta.internal.metals	[121,3489,576]
patoh	1	100	scala.meta.internal.metals.ammonite	[0,75,0]
patoh	1	100	scala.meta.internal.metals.codeactions	[0,104,0]
patoh	1	100	scala.meta.internal.metals.code lenses	[0,64,0]
patoh	1	100	scala.meta.internal.metals.config	[0,26,0]
patoh	1	69	scala.meta.internal.metals.debug	[18,499,200]
patoh	1	100	scala.meta.internal.metals.newScalaFile	[0,82,0]
patoh	1	57	scala.meta.internal.mtags	[259,472,85]
patoh	1	65	scala.meta.internal.parsing	[0,135,71]
patoh	1	100	scala.meta.internal.process	[0,37,0]
patoh	1	100	scala.meta.internal.remotels	[0,30,0]
patoh	1	100	scala.meta.internal.rename	[0,86,0]
patoh	1	100	scala.meta.internal.semver	[0,24,0]
patoh	1	100	scala.meta.internal.traubleshoot	[0,49,0]
patoh	1	73	scala.meta.internal.tvp	[0,236,87]
patoh	1	100	scala.meta.internal.worksheets	[0,173,0]
patoh	1	100	scala.meta.metals	[0,12,0]
patoh	1	70	tests	[6,802,323]
patoh	1	100	tests.codeactions	[0,21,0]
patoh	1	47	tests.pc	[5,11,7]
patoh	2	93	bench	[0,10,140]
patoh	2	100	bill	[0,0,67]
patoh	2	90	scala.meta.internal.pc	[12,78,815]
patoh	2	100	scala.meta.internal.pc.completions	[0,0,394]
patoh	2	100	scala.tools.nsc.interactive	[0,0,18]

Table 8.4: Assignment of packages of the *metals* project to different partitions (0, 1, 2) for a 3-way partitioning computed with *patoh* using the SCG graph. The package partition is determined by the dominant partition number among code entities in that package. Packages with low accuracy (high spread of entities among different partitions) are the ‘contact points’ between different partitions and might need to be reorganized before modularization of the project.

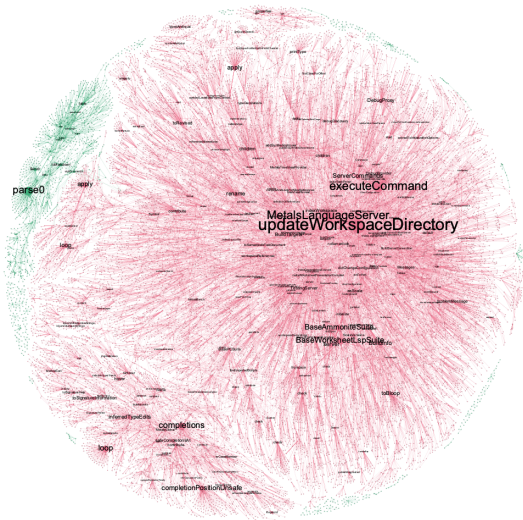
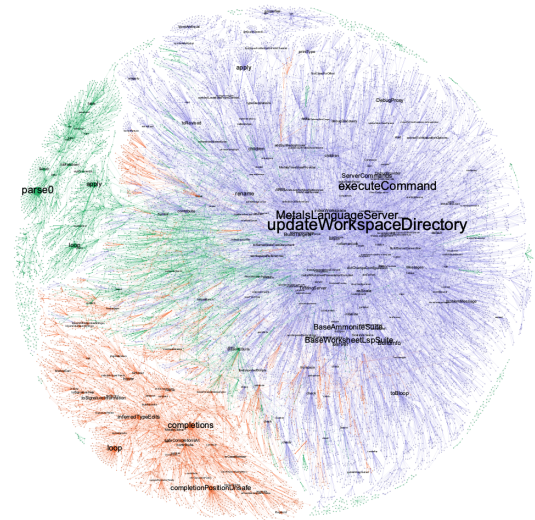
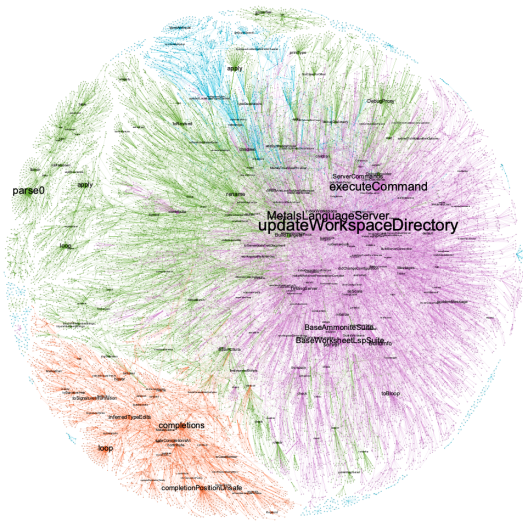
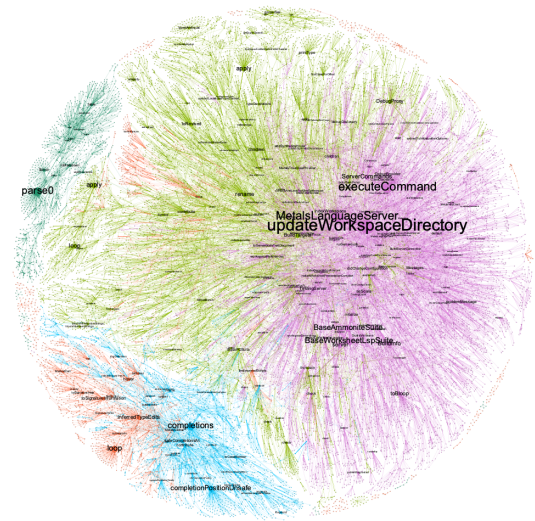
(a) SCG *metals* project; *gpmetis*; 2 partitions(b) SCG *metals* project; *gpmetis*; 3 partitions(c) SCG *metals* project; *gpmetis*; 4 partitions(d) SCG *metals* project; *gpmetis*; 5 partitions

Figure 8.8: Visualization [40] of the *metals* project SCG graph, for 2-5 partitions, using *gpmetis* with high partition size imbalance allowed. High-degree nodes are labeled.

However, the visualization of the SCG graph is difficult to comprehend due to a large number of nodes and edges. For comparison, the corresponding visualization of the CCN graph is presented in Fig. 8.9. Having looked at this visualization and upon further investigation of the package assignment to partitions, we can conclude that the *metals* project architecture is hard to modularize. There is an evident tight coupling between entities and the biggest part of the software, cluttered in the center of the graph, contains big hubs such as *MetalsLanguageClient*, *MetalsLanguageClient* or *BuildTargets* which represents the bulk functionality of the LSP server. Some smaller and better separated modules are devoted to documentation generation (e.g., *ScaladocParser*) or symbol utilities (e.g., *Symbol*). We have confirmed our findings directly with the current maintainer of the *metals* project.

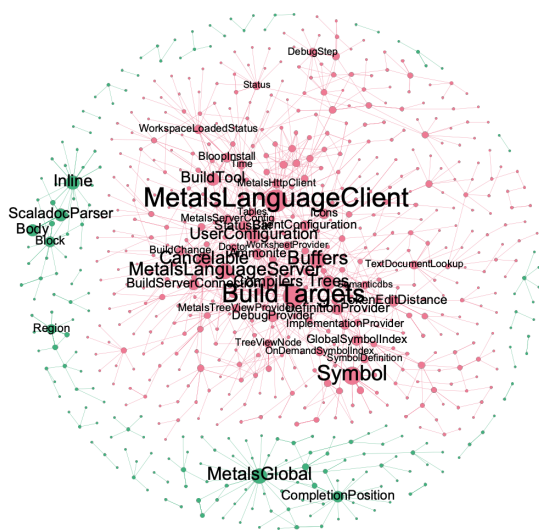
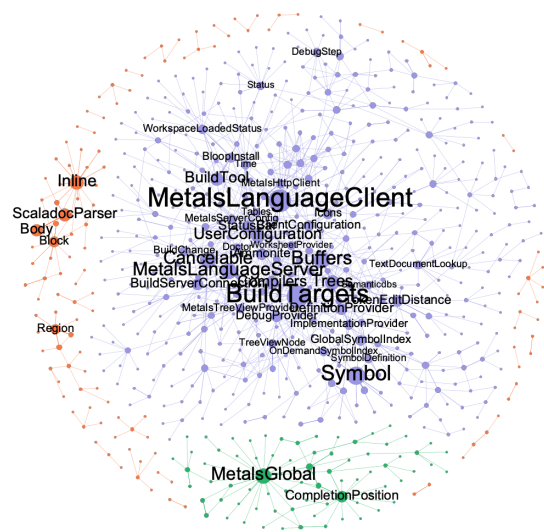
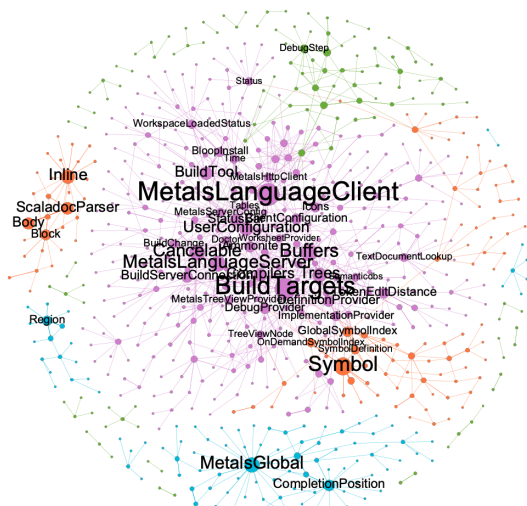
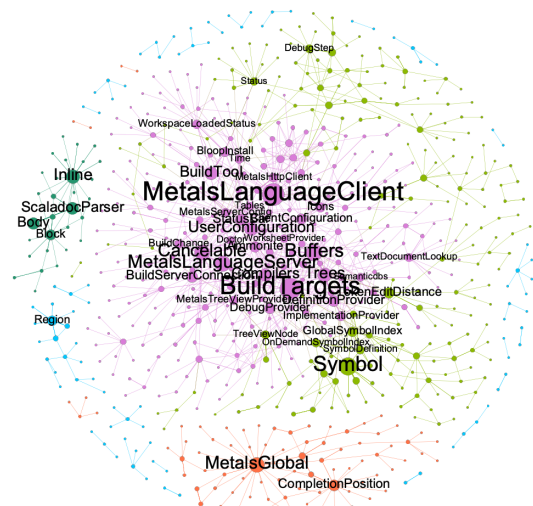
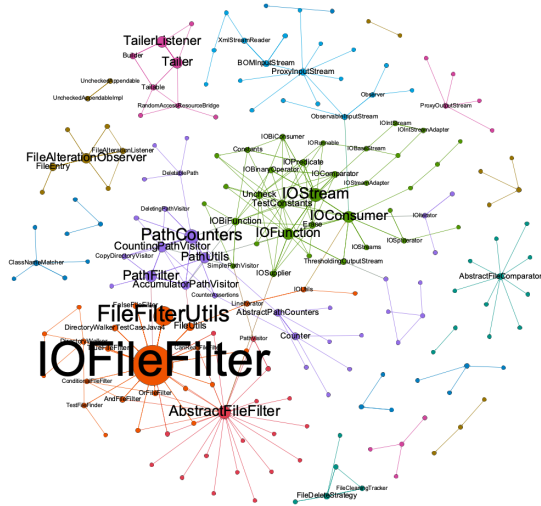
(a) CCN *metals* project; gpmets; 2 partitions(b) CCN *metals* project; gpmets; 3 partitions(c) CCN *metals* project; gpmets; 4 partitions(d) CCN *metals* project; gpmets; 5 partitions

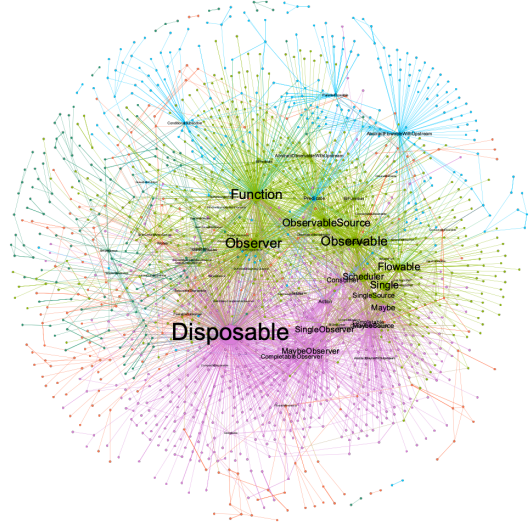
Figure 8.9: CCN visualization [40] of the *metals* project, for 2-5 partitions, using *gpmets* with high partition size imbalance allowed. High-degree nodes are labeled.

To compare our findings for the *metals* project with other cases, we have partitioned and visualized in a similar manner the CCN networks for four other projects: *commons-io*, *RxJava*, *akka*, and *spark*. The results are presented in Fig. 8.10. We have computed the partitioning using *gpmetis*, allowing for maximum imbalance. Contrary to *metals*, in the case of these four projects, the discovered architecture seems to exhibit a highly modular structure, with no single partition dominating the project as observed in the *metals* case. This suggests that the *metals* project should indeed be considered an anomaly. For the *metals* project, if a split of the major partition were required, some refactoring would be necessary beforehand.

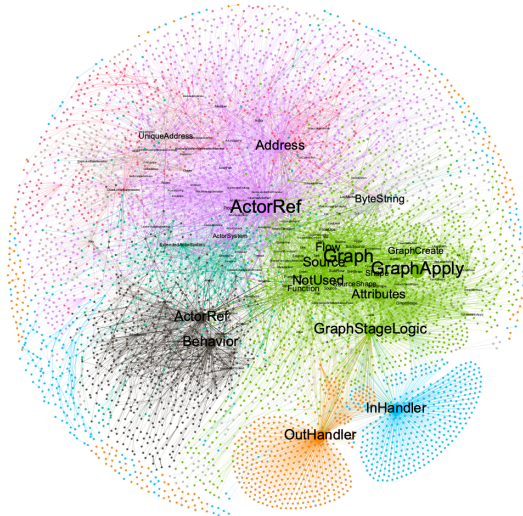
To examine the architecture of the *metals* software from the perspective of software execution, we can partition and analyze the CG network, as depicted in Fig. 8.11. It is evident that, once again, there is one dominant partition that aligns with the architecture recovered from the CCN. In the CG network, from the standpoint of software execution, the betweenness centrality — which represents the number of shortest paths passing through a given node — is a critical property. It indicates methods frequently involved in the call chain, and as such, these nodes occupy a central position in the extracted software execution architecture.



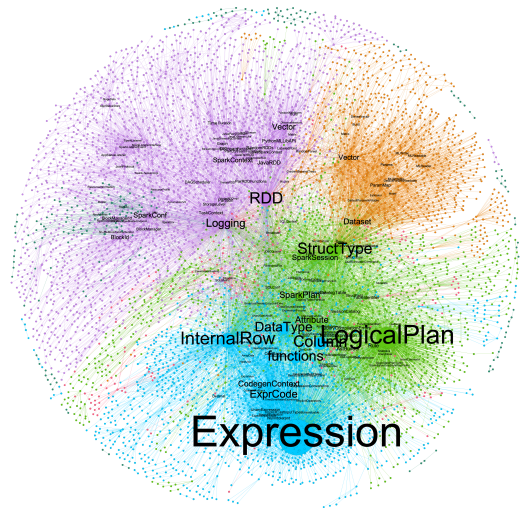
(a) CCN *commons-io* project. 174 nodes, 438 edges, 7 partitions (*gpmetis*)



(b) CCN *RxJava* project. 1564 nodes, 6787 edges, 6 partitions (*gpmetis*)



(c) CCN *akka* project (without tests). 5477 nodes, 21254 edges, 9 partitions (*gpmetis*)



(d) CCN *spark* project. 6461 nodes, 31742 edges, 6 partitions (*gpmetis*)

Figure 8.10: CCN visualization [40] of four projects: *commons-io*, *RxJava*, *akka*, *spark*. Colors denote different partitions computed with *gpmetis*, with maximum partition imbalance allowed. For all projects we can visually localize clearly separated parts and no single partition that dominates the entire project. High-degree nodes are labeled.

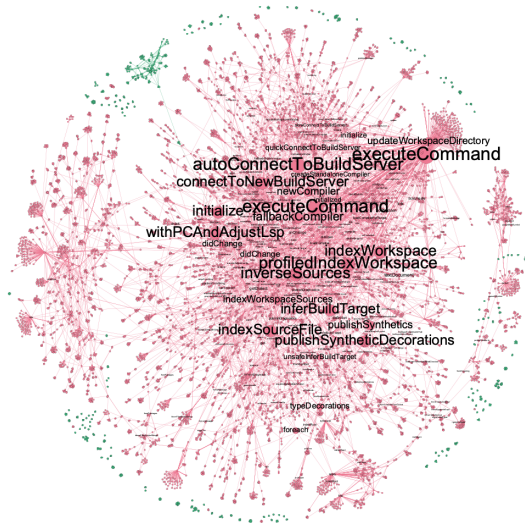
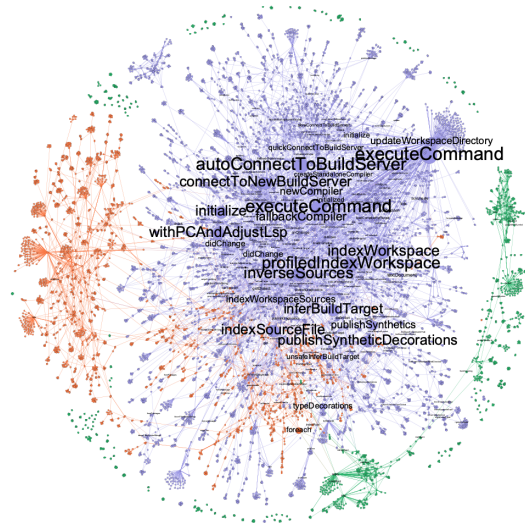
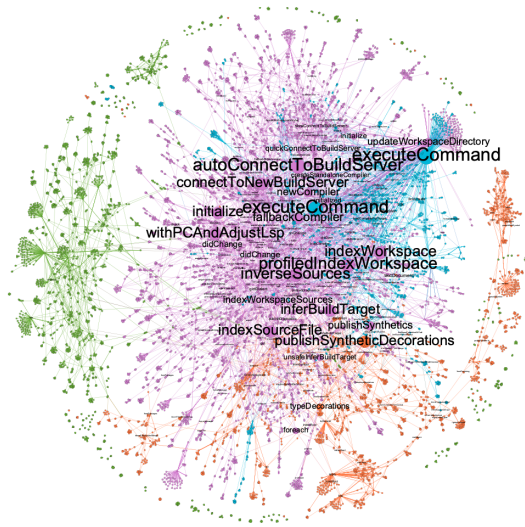
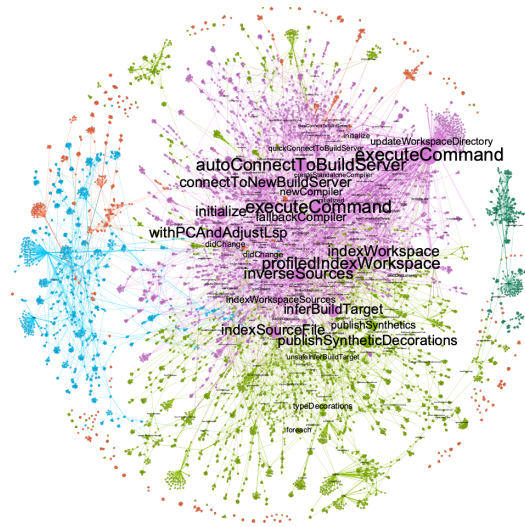
(a) CG *metals* project; *gpmets*; 2 partitions(b) CG *metals* project; *gpmets*; 3 partitions(c) CG *metals* project; *gpmets*; 4 partitions(d) CG *metals* project; *gpmets*; 5 partitions

Figure 8.11: CG graph visualization [51] of the *metals* project, 2-5 partitions, using *gpmets* with high partition size imbalance allowed. Nodes with a high Betweenness Centrality are labeled.

8.5 Discovering critical project entities

The most important nodes in the project are usually the starting points that developers look at to comprehend the project completely from the beginning. An ordered list of the most influential entities can help with a more structured approach to reading the source code and learning about the project. Moreover, the most important nodes are those with the largest *impact of change* [67]. In this case, the Semantic Code Graph representation, which is effectively the result of a static structural dependence program analysis, can act as an intermediate program representation used in the Change Impact Analysis [66]. Due to a potential high impact of change, when adjusting the source code of the top entities during software maintenance, the code review process should be conducted thoroughly with extra care to preserve high source code quality for the most critical parts of the system.

We can assess the importance of project entities in a project by measuring the node size (in terms of the number of lines of code or by its outgoing/incoming degree), or by taking into account different metrics known in graph theory, such as graph centrality measures or node influence measures [43]. For all nodes in the SCG graphs of the studied projects, we have computed nine different properties that can serve as different metrics to determine the top nodes:

1. LOC – top nodes by Lines-of-Code metrics. Code entities that span a significant number of lines of code naturally have a greater impact on the project.
2. Outgoing Degree – top nodes by the number of outgoing edges. Nodes with high degree suggest code entities aggregating big and likely important functionalities.
3. Incoming Degree – top nodes by the number of incoming edges. Many incoming edges denote extensive direct usage of this node throughout the project.
4. Eigenvector Centrality – top nodes computed by the Eigenvector Centrality [18] algorithm. High value means that a node is connected to multiple nodes that themselves have a high EC score. This metric focuses on direct connections.
5. Katz Centrality – top nodes computed by the Katz Centrality [58] algorithm. High value of Katz Centrality metric is influenced not only by direct neighbors of a node but also by nodes connected indirectly.
6. Page Rank – top nodes computed by Page Rank algorithm [24]. Page Rank centrality is a variant of Eigenvector Centrality which assigns a PR value to a node based on the number of nodes linking to this node and their PR values. This metric is affected by the importance of incoming links from highly ranked nodes.
7. Betweenness Centrality – top nodes computed by the Betweenness Centrality algorithm [22]. High value of BC denotes that there are multiple shortest paths between other nodes going through this node. This metric indicates nodes crucial for the flow of information in the network.

8. Harmonic Centrality – top nodes computed by the Harmonic Centrality algorithm [94] (Closeness Centrality metric for disconnected graphs). A node with high value of Harmonic Centrality is one that is close to many other nodes in the network.

Finally, we propose a combined importance metric that takes into account all metrics described above:

9. Combined importance – the metric is computed as follows: for each metric 1-8, take 20 nodes with the highest value (top 20 nodes). Give 1 point to each node which is within top 20 nodes for a given metric. Rank the nodes based on the sum of their scores. In other words, the top nodes will be the ones that scored highly in multiple metrics.

The most important nodes in the full SCG, CCN and the CG in terms of different centrality metrics are presented in the Tables 8.5, 8.6, and 8.7. It should be noted that the resulting top nodes vary significantly for different metrics. In fact, metrics capture different aspects of potential node importance [43]. For example, high out-degree points to hubs, which can represent a structure encapsulating bigger chunk of related functionalities (such as class nodes), whereas high in-degree reveals nodes which are referenced by many other nodes and can be treated as project main building blocks. Metrics such as Eigenvector, Katz, or PageRank Centrality measure the importance of the node, which is influenced by the importance of its neighbors. Such nodes can be the most important code units from the maintenance point of view, where the introduced changes can have a great project-wide impact. Metrics such as Betweenness Centrality or Harmonic Centrality take into account the topological position in the graph and show the distance-wise central nodes through which many shortest paths lead, or nodes which are the closest nodes to multiple other nodes in the graph. These metrics might be especially meaningful for the Call Graph, indicating frequently called nodes on different execution paths.

Finally, there are nodes ranked highly according to multiple metrics, indicating units particularly important for the entire project. Our *combined importance* metric was proposed with the purpose of finding such cases. The top three nodes according to the *combined importance* metric for each project are presented in Tab. 8.8. It is particularly interesting to observe the difference in results obtained from the Full SCG versus the Call Graph and Class Collaboration Network. For example, the critical SCG nodes for the *metals* project are classes *MetalsLanguageServer*, *BuildTargets*, and *MetalsLanguageServer#languageClient*. These nodes are important from the *project structure perspective*, i.e., for the purpose of software comprehension in the context of project understanding or planning refactoring activities. Indeed, *metals* is a Language Server Protocol implementation for Scala, where *MetalsLanguageServer#languageClient* represents the tool (IDE) and *MetalsLanguageServer* implements all the functionalities required by the client, so these are two most important and complementary classes. *BuildTargets* is a non-obvious candidate for the important node, but on the closer inspection we can learn that *BuildTargets* is a in-memory cache for looking up build server metadata and is involved in most of the *metals* functionalities. For the Class Collaboration Network, *BuildTargets* is also the most important node, followed by *Cancelable* and *Compilers*. *Cancelable* is a trait that is extended by more than twenty different classes, making it an important node from the class inheritance perspective. The *Compilers* class is used in eight different places as a constructor argument for eight top-level classes in *Metals*, and because of that, it has a high influence score (it is a top node according

	1. Lines of Code	#	2. Outgoing Degree	#	3. Incoming Degree	#
Semantic Code Graph						
retrofit	RequestFactory	800	parseParameterAnnotation	674	Call	147
commons-io	IOUtils	3608	FileUtils	176	IOFileFilter	131
playframework	Multipart	706	PlayDocsValidation	367	Mapping	882
metals	MetalsLanguageServer	2501	updateWorkspaceDirectory	503	server	157
glide	BaseRequestOptions	1398	initializeDefaults	261	with	183
vert.x	HttpClientOptions	1448	MimeMapping	990	MimeMapping#m	989
RxJava	Flowable	20738	Flowable	541	Disposable	1251
dubbo	URL	1714	PojoUtils#realize1	390	URL	1891
spring-boot	ServerProperties	1757	configureProperties	300	from	330
akka	Source	3896	ByteStringSpec	2031	ActorRef#!	2747
spark	functions	5380	ScalaUDF	2677	Expression	3561
Class Collaboration Network						
retrofit	RequestFactory	800	HttpServiceMethod	12	Converter	115
commons-io	IOUtils	3608	FileFilterUtils	48	IOFileFilter	102
playframework	Multipart	706	Forms	582	Mapping	876
metals	MetalsLanguageServer	2501	MetalsLanguageServer	29	MetalsLanguageClient	37
glide	BaseRequestOptions	1398	Engine	60	Options	143
vert.x	HttpClientOptions	1448	FileSystemImpl	110	Handler	615
RxJava	Flowable	20738	Observable	317	Disposable	927
dubbo	URL	1714	RegistryProtocol	90	URL	1250
spring-boot	ServerProperties	1757	Binder	78	ConfigurationPropertyName	150
akka	Source	3896	GraphApply	853	ActorRef	1441
spark	functions	5380	functions	851	Expression	2922
Call Graph						
retrofit	parseParameterAnnotation	456	parseParameterAnnotation	417	Builder#method	77
commons-io	FilenameUtils#doNormalize	96	FilenameUtils#doNormalize	119	CloseableURLConnection#urlConnection	46
playframework	webSocketProtocol	349	forwardsRouter#apply	120	unbindAndValidate	257
metals	updateWorkspaceDirectory	419	updateWorkspaceDirectory	431	XtensionJavaFuture#asScala	101
glide	initializeDefaults	245	initializeDefaults	183	Glide#with	182
vert.x	EventBusOptionsConverter#fromJson	254	TCPServerBase#listen	170	Future#onComplete	170
RxJava	MemoryPerf#main	378	ConcatMapEagerMainObserver#drain	134	Exceptions#throwIfFatal	347
dubbo	PojoUtils#realize1	249	PojoUtils#realize1	229	StringUtils#isEmpty	418
spring-boot	MavenBuild#execute	73	configureProperties	233	PropertyMapper#from	329
akka	UnzipWith22#createLogic	476	DistributedPubSubMediator#receive	256	GraphStageLogic#pull	1501
spark	prepareSubmitEnvironment	641	prepareSubmitEnvironment	679	Params#\$	1382

Table 8.5: Top nodes in the studied projects in terms of general properties (1. Lines of Code, 2. Outgoing Degree, 3. Incoming Degree);

to Eigenvector Centrality and Page Rank metrics). On the other hand, the results obtained from the Call Graph clearly reflect *the runtime perspective*, focusing on invocation dependencies. For the *metals* project, the important nodes in the Call Graph are the *MetalsLanguageServer#executeCommand* method which is responsible for executing a large set of commands, the *MetalsLanguageServer#updateWorkspaceDirectory* method responsible for initialization of almost forty different providers in *MetalsLanguageServer*, and the *XtensionJavaFuture#asScala* helper method used in all the places when *metals* interacts directly with the underlying Java based LSP4J³ client. These methods are critical from the runtime, and hence performance and program execution correctness, perspective.

³<https://projects.eclipse.org/projects/technology.lsp4j>

	4. Eigenvector Centrality	#	5. Katz Centrality	#	6. Page Rank	#
Semantic Code Graph						
retrofit	Utils#toResolve	0.6051	Call	2.5066	Call	0.0180
commons-io	Tailer#tailable	0.2603	IOFileFilter	2.3425	IOFileFilter	0.0131
playframework	Request#[A]	0.2977	Mapping	10.1967	Mapping	0.0105
metals	Version#major	0.4237	BaseLspSuite#server	2.5840	AnsiStateMachine#apply	0.0038
glide	TranscodeType	0.3777	Glide#with	2.8629	Key	0.0069
vert.x	HttpMethod	0.4860	MimeMapping#m	10.8902	HttpMethod	0.0143
RxJava	Flowable	0.5522	Disposable	13.9548	Disposable	0.0370
dubbo	URL#urlAddress	0.3399	URL	20.9279	URL	0.0201
spring-boot	Regex#group	0.5066	PropertyMapper#from	4.3904	DependencyVersion	0.0050
akka	Source	0.4295	ActorRef#	29.0573	ReplicatedData	0.0246
spark	ConfigBuilder#version	0.3444	Expression	37.9700	Expression	0.0158
Class Collaboration Network						
retrofit	Converter	0.8521	Converter	2.1933	Callback	0.2146
commons-io	TailerListener	0.8256	IOFileFilter	2.0228	PathFilter	0.0538
playframework	Binding	0.9392	Mapping	9.7610	TypedEntry	0.3800
metals	Cancelable	0.5952	MetalsLanguageClient	1.3770	Cancelable	0.0174
glide	Transformation	0.3522	Options	2.4766	Editor	0.0605
vert.x	Handler	0.7364	Handler	8.3021	Buffer	0.1275
RxJava	Scheduler	0.6571	Disposable	10.7265	ParallelFlowable	0.1242
dubbo	ApplicationModel	0.4460	URL	13.9248	URLAddress	0.2983
spring-boot	ConditionMessage	0.7171	ConfigurationPropertyName	2.5881	Builder	0.2276
akka	ForwardOps	0.4815	Graph	16.1818	ActorRef	0.1371
spark	Column	0.5824	Expression	31.2843	Param	0.1191
Call Graph						
retrofit	Utils#resolve	0.4450	Builder#method	1.7843	Utils#getRawType	0.0124
commons-io	FileAlterationObserver#listFiles	0.8323	CloseableURLConnection#urlConnection	1.4605	AbstractFileFilter#accept	0.0159
playframework	Multipart#partStart	0.3389	Mapping#bind	3.5704	Configuration#underlying	0.0018
metals	SemanticdbTreePrinter#printSymbol	0.4449	XtensionJavaFuture#asScala	2.0247	XtensionAbsolutePath#path	0.0028
glide	RequestBuilder#thumbnailBuilder	0.4773	Glide#with	2.8308	Preconditions#checkNotNull	0.0040
vert.x	JsonParserImpl#handleEvent	0.4855	Future#onComplete	2.7083	JsonObject#map	0.0065
RxJava	MergeObserver#observers	0.5741	Exceptions#throwIfFatal	4.4854	DISPOSED	0.0040
dubbo	JavaBeanDescriptor	0.5246	StringUtils#isEmpty	5.3694	URL#getParameter	0.0068
spring-boot	JSONTokenizer#pos	0.7939	PropertyMapper#from	4.3362	PropertiesConfigAdapter#properties	0.0012
akka	WorkPullingProducerControllerImpl#context	0.3596	GraphStageLogic#pull	16.4494	ActorRef#	0.0021
spark	StructField#name	0.8183	Params#\$	15.0390	AnalysisException#<init>	0.0030

Table 8.6: Top nodes in the studied projects in terms of influence-based centrality metrics (4. Eigenvector Centrality, 5. Katz Centrality, 6. Page Rank);

	7. Betweenness Centrality*	#	8. Harmonic Centrality	#	9. Combined importance	#
Semantic Code Graph						
retrofit	Retrofit	537k	Builder#parseParameterAnnotation	0.1023	Retrofit	5
commons-io	IOStreams#forAll	301k	FileUtils	0.0924	IOConsumer	4
playframework	RequestHeader	3 434k	Forms	0.0622	RequestHeader	5
metals	Ammonite	6 789k	MetalsLanguageServer	0.1951	MetalsLanguageServer	4
glide	Glide	7 762k	Glide	0.0854	RequestBuilder	6
vert.x	VertxImpl	45 953k	VertxImpl	0.1208	VertxImpl	4
RxJava	Flowable	18 802k	Flowable	0.0673	Flowable	7
dubbo	URL	79 424k	DubboBootstrap	0.0450	URL	7
spring-boot	Binder	1 107k	BuildImageMojo#buildImage	0.0106	ConfigurationPropertyName	5
akka	TraversalBuilder	106 628k	SystemMessageDeliverySpec	0.0526	ActorRef	3
spark	SparkContext	714 504k	BaseSessionStateBuilder	0.1034	Expression	4
Class Collaboration Network						
retrofit	Retrofit	0.243k	HttpServiceMethod	0.0490	Retrofit	8
commons-io	AbstractFileFilter	0.075k	Uncheck	0.0636	IOFileFilter	4
playframework	Result	2.4k	Helpers	0.0377	RequestHeader	5
metals	BuildTargets	1.2k	DebugProvider	0.0773	BuildTargets	4
glide	Glide	6.0k	Glide	0.0827	RequestManager	5
vert.x	Vertx	13.7k	VertxImpl	0.1507	JsonObject	4
RxJava	Flowable	12.5k	Single	0.0197	Flowable	8
dubbo	ApplicationModel	51.0k	DubboBootstrap	0.0355	ApplicationModel	5
spring-boot	SpringApplication	1.2k	Builder	0.0095	ConfigurationPropertyName	4
akka	Props	2.7k	UnzipWithApply	0.0084	ActorRef	4
spark	SparkContext	256.1k	SparkSession	0.0200	Expression	5
Call Graph						
retrofit	Retrofit#create	15k	Builder#parseParameterAnnotation	0.0895	Builder#parseParameterAnnotation	4
commons-io	FilenameUtils#getPrefixLength	2k	XmlStreamReader	0.0131	FilenameUtils#doNormalize	4
playframework	RangeResult#ofSource	1k	PlayRequestHandler#handle	0.0127	Configuration#get	3
metals	MetalsLanguageServer#executeCommand	35k	MetalsLanguageServer#executeCommand	0.0627	MetalsLanguageServer#executeCommand	4
glide	Glide#initializeGlide	107k	MultiRequestTest#(...)whenRequestListenersCalled	0.0167	Glide#with	4
vert.x	TCPSSLOptions	16k	BareCommand#startVertx	0.0108	ContextInternal#promise	3
RxJava	SpseLinkedListArrayQueue	1690	MemoryPerf#main	0.0028	DisposableHelper#dispose	4
dubbo	FrameworkModel#defaultModel	1 408k	ReferenceConfig#init	0.0113	Node#getUrl	3
spring-boot	SpringApplication#run	30k	BuildImageMojo#buildImage	0.0061	SpringApplication#run	4
akka	GraphStageLogic#internalCompleteStage	137k	ClusterCoreDaemon#initialized()	0.0034	GraphStageLogic#pull	4
spark	SparkContext#clean	242k	ALSEExample#main	0.0052	Params#set	3

Table 8.7: Top nodes in the studied projects in terms of distance-based centrality metrics (7. Betweenness Centrality, 8. Harmonic Centrality) and according to the proposed 9. *combined metric*. *the value of Betweenness Centrality was not normalized and the total number of shortest paths going through a particular node is used for more convenient result presentation.

Name	#1	#	#2	#	#3	#
Semantic Code Graph						
retrofit	Retrofit	5	parseParameterAnnotation	4	Converter	4
commons-io	IOConsumer	4	IOCase	3	IOStream	3
playframework	RequestHeader	5	Result	4	AkkaHttpServer	3
metals	MetalsLanguageServer	4	BuildTargets	2	MetalsLanguageServer#languageClient	2
glide	RequestBuilder	7	RequestManager	4	Request	3
vert.x	VertxImpl	4	ContextInternal	4	Future	4
RxJava	Flowable	7	Observable	6	Single	5
dubbo	URL	7	ApplicationModel	4	Invoker	3
spring-boot	ConfigurationPropertyName	5	SpringApplication	3	ConfigurationPropertySource	3
akka	ActorRef	3	Source	3	ActorRef#!	2
spark	Expression	4	SparkContext	3	RDD	3
Class Collaboration Network						
retrofit	Retrofit	8	Factory	6	Call	5
commons-io	IOFileFilter	4	IOStream	4	AbstractFileFilter	4
playframework	RequestHeader	5	Result	5	PlayBodyParsers	4
metals	BuildTargets	4	Cancelable	4	Compilers	4
glide	RequestManager	5	RequestBuilder	5	Key	4
vert.x	JsonObject	4	Vertx	4	Handler	4
RxJava	Flowable	8	Single	6	Maybe	5
dubbo	ApplicationModel	5	URL	5	ModuleModel	5
spring-boot	ConfigurationPropertyName	4	ServerProperties	4	ConditionOutcome	4
akka	ActorRef	4	SubFlow	3	GraphStageLogic	3
spark	Expression	5	DataType	4	RDD	4
Call Graph						
retrofit	parseParameterAnnotation	4	Utils#resolve	4	RequestFactory.Builder#build	4
commons-io	FilenameUtils#doNormalize	4	XmlStreamReader#calculateHttpEncoding	3	IOBaseStream#unwrap	3
playframework	Configuration#get	3	RequestHeader#attrs	3	Route#call	3
metals	MetalsLanguageServer#executeCommand	4	updateWorkspaceDirectory	3	XtensionJavaFuture#asScala	3
glide	Glide#with	4	Preconditions#checkNotNull	3	LoadBytesTest#context	3
vert.x	TCPServerBase#listen	3	ContextInternal#promise	3	BufferImpl#buffer	3
RxJava	DisposableHelper#dispose	4	ConcatMapEagerMainObserver#drain	4	DisposableHelper#setOnce	3
dubbo	Node#getUrl	3	DubboCodec#decodeBody	3	StringUtils#isEmpty	3
spring-boot	SpringApplication#run	4	PropertyMapper#get	3	FlywayConfiguration#configureProperties	3
akka	GraphStageLogic#pull	4	GraphStageLogic#completeStage	4	ActorRef#!	3
spark	Column#expr	3	Params#set	3	Expression#dataType	3

Table 8.8: Top 3 nodes in the studied projects according to the **combined importance** calculated for the full SCG, the Class Collaboration Network, and the Call Graph.

8.6 Finding code similarities

Finding code similarities in the project can help both in software comprehension and directly in software maintenance. To better understand the software, we can look for cases of similar code in the project in order to discover not obvious relations and analyze subtle differences between them [77]. In software maintenance, we use clone detection to find and refactor code duplication [39] or to apply bug fixes in multiple places. It has been proven that graph based representation of the source code (such as Program Dependence Graph [38]) can be used to find code similarities [59, 61]. Using the SCG graph, we can find *semantically similar* code fragments within the project. Existing algorithms are typically based on computing graph similarity [59, 61, 97]. However, the disadvantage of this approach is its high computational complexity and, consequently, slow performance for large code bases. We propose a simple and effective heuristic for quick finding similarity between methods in a given project using the SCG. The heuristic, fully presented in Listing 8.6, can be summarized as follows:

- **Step 1.** Generate a list of all method pairs in the project.
- **Step 2.** For each method in a pair, retrieve a set of its adjacent nodes; then compute the intersection of the sets.
- **Step 3.** Determine the similarity between methods based on the size of their intersection set. Two nodes are treated as similar if for both nodes the minimal size of the intersection is higher than `s_min` and the fraction of the intersecting nodes (within all adjacent nodes) is higher than `s_min_p`.
- **Step 4.** Discard similar methods having the same parent to exclude similarities occurring in the same code unit and focus on non obvious ones.

Listing 8.6: Algorithm for finding similar methods in a project.

```
import itertools as it
import scg

# min. 5 nodes in intersection
s_min = 5
# min. 50% of method nodes has to be the same
s_min_p = 50
scg_files = scg.read_scg("data/metals")
G = scg.create_graph(scg_files)
# find all method nodes

# Step 1.
methods = [
    n
    for n in G.nodes()
    if G.nodes[n].get("scg_node").kind == "METHOD"
]
# generate all method pairs
```

```

methods_comb = list(it.combinations(methods, 2))

# find defining node for given node n
def parent(G, n):
    return [
        p
        for p in G.predecessors(n)
        if G[p][n]["type"] == "DECLARATION"
    ]

similar = []
for (
    n1,
    n2,
) in methods_comb:
    # Step 2.
    n1_n = set(G.successors(n1))
    n2_n = set(G.successors(n2))
    s = len(set(n1_n).intersection(n2_n))
    # fraction of common nodes in each method
    n1_p = 0 if s==0 else int(s / len(n1_n) * 100)
    n2_p = 0 if s==0 else int(s / len(n2_n) * 100)
    # Step 3.
    if s >= s_min and (
        n1_p >= s_min_p and n2_p >= s_min_p
    ):
        # Step 4.
        if parent(G, n1) != parent(G, n2):
            similar.append((n1, n2, s, n1_s, n2_s))
# 'similar' contains pairs of similar methods
print(similar)

```

Even such a simplified heuristic can reveal interesting similarities, for example in the *metals* project, as shown in Fig. 8.12. Two similar methods –*oldReloadResult* and *oldInstallResult* – were found which could be refactored into a *common* method presented in Listing 8.9.

This example demonstrates how code analysis algorithms can be implemented using the raw SCG graph data structures. Thanks to the rich dependency model of the SCG, we were able to easily adjust the similarity search algorithm and discard results found within same parent in order to focus on less obvious comparisons. More complex algorithms for finding code similarities can be implemented in a similar fashion, however, it is out of scope of this dissertation.

Listing 8.7: *metals* method from *WorkspaceReload* class.

```

def oldReloadResult(
  digest: String
): Option[WorkspaceLoadedStatus] = {
  if (tables.dismissedNotifications.ImportChanges.isDismissed) {
    Some(WorkspaceLoadedStatus.Dismissed)
  } else {
    tables.digests.last().collect {
      case Digest(md5, status, _) if md5 == digest =>
        WorkspaceLoadedStatus.Duplicate(status)
    }
  }
}

```

Listing 8.8: *metals* method from *BloopInstall* class.

```

private def oldInstallResult(
  digest: String
): Option[WorkspaceLoadedStatus] = {
  if (notification.isDismissed) {
    Some(WorkspaceLoadedStatus.Dismissed)
  } else {
    tables.digests.last().collect {
      case Digest(md5, status, _) if md5 == digest =>
        WorkspaceLoadedStatus.Duplicate(status)
    }
  }
}

```

Listing 8.9: Extracted common code from *oldReloadResult* and *oldInstallResult* methods.

```

private def common(
  digest: String,
  condition: Boolean,
  tables: Tables
): Option[WorkspaceLoadedStatus] = {
  if (condition) {
    Some(WorkspaceLoadedStatus.Dismissed)
  } else {
    tables.digests.last().collect {
      case Digest(md5, status, _) if md5 == digest =>
        WorkspaceLoadedStatus.Duplicate(status)
    }
  }
}

```

Figure 8.12: Method *oldReloadResult* and *oldReloadResult* from *metals* project are similar, differing only in naming and the first condition statement. Consequently, a *common* method shown in Listing 8.9 can be used to replace them.

Chapter 9

Summary of results

9.1 Answering research questions

Answer to RQ1: Does the SCG model enhance software comprehension capabilities in comparison with the Class Collaboration Network and the Call Graph models?

In addressing RQ1, empirical studies confirm the SCG model’s superior capabilities in software comprehension. However, we have found that the Class Collaboration Network and the Call Graph models also provide valuable perspectives.

In Chapter 6 we have shown that unique properties of the SCG model (such as the required *location* property) allow interactive source browsing in the full context of detailed dependencies and code entities that are neither with standard Call Graph analysis nor Class Collaboration Network. The closeness to the source code and the support for the protobuf storage format enable efficient and detailed visualization. Through the creation of the *Graph Buddy* plugin and its evaluation we prove that the SCG model is a superior model for source code dependency comprehension through interactive visualization.

In the case of project structure comprehension, we have combined the computation of project metrics with data analysis based on the split-apply-combine approach [120]. The richness of the full SCG model allowed us to discover interesting properties of the analyzed software which could not have been found based on the Call Graph or Class Collaboration Network alone. For example, we have discovered excessive use of variables in the *spring-boot* project and localized the relevant places in the project (Section 8.2). Additionally, we have shown a power law distribution of node degrees in software projects and located outliers, i.e. unusually highly connected code entities. Only thanks to the richness of the SCG model, we could identify all unique hub outliers for both classes (e.g., *ActorRef*) and methods (e.g., *ActorRef#!*).

In the experiment to identify top important code entities, we have shown that the full SCG model, the Class Collaboration Network, and the Call Graph model provide different interesting perspectives on the project structure. Each perspectives is useful and, notably, the SCG model encompasses all of them, being a superset of both CCN and CG.

Answer to RQ2: Does SCG-based data analysis enable actionable software comprehension insights?

Several project comprehension activities presented earlier led to useful insights regarding code problems and possible solutions. In Section 8.2 we discovered an excessive number of variables in the *spring-boot* project. A closer look into the source code allowed us to understand the issue and propose the refactoring exercise to apply *final* modifier to reduce accidental mutability and in consequence improve the code readability.

In Section 8.3, which focuses on project structure comprehension, we identified an anomaly in the *Retrofit* project. Specifically, the *parseParameterAnnotation* method exhibited an unusually high outgoing degree. We conducted an analysis to understand the cause of this anomaly and proposed refactoring activities.

Finally, in Section 8.6, we were able to find similar methods in different places in the code, which is not only relevant for software comprehension, but also led to proposing an improvement of the source code by extracting the *common* method (presented in Listing 8.9) and removing code duplication.

9.2 Impact of results

We believe that the presented results of our research indicate that the Semantic Code Graph (SCG) represents a significant development in the field of code dependency analysis and software comprehension. For researchers and experts, this work carries several benefits:

1. **Comprehensive code dependency analysis:** SCG offers a detailed source code information model providing insight into code structure, including code definitions, declarations, and their relationships. This offers significant depth of analysis which opens up new avenues for understanding software systems. We have demonstrated the usefulness of the SCG model in diverse software comprehension applications, including code structure visualization, identification of crucial software entities, investigation of code quality issues, and software mining using data analytics.
2. **Practical utility:** The empirical evaluation of the SCG on real-world projects demonstrates its practical utility. This utility has been demonstrated by using SCG as foundation of several tools for software comprehension: the *Graph Buddy* plugin for interactive code dependency visualization in popular IDEs, and the *scg-cli* tool. Researchers can use the tools that we have provided to conduct similar experiments for the purpose of software comprehension or validation of their own models. This emphasis on empirical validation helps bridge the gap between theoretical work and real-world applications on software projects.
3. **Actionable insights:** We have shown that the SCG capability has led to concrete useful insights, e.g., identification of excessive usage of mutable variables, identification of outliers with extreme number of dependencies, or finding similar methods and suggesting project refactoring actions. Researchers can build on this capability to develop automated code quality improvement tools. SCG's high level of detail enables deriving other graph-based code representation models, such as Class Collaboration Network (CCN) and Call Graph (CG), which are already provided by the *scg-cli* tool. Our experi-

ments have shown that these models provide additional insights into software structure and execution dependencies.

4. **External integrations:** Along with the SCG model we have provided its portable intermediate representation, tools and APIs that enable integrations with external tools such as Gephi, Jupyter Notebook or any tool capable of reading a protobuf format.
5. **Community contribution and future research:** SCG detailed description and protobuf based intermediate representation facilitates creation of new extractors for other programming languages and the *scg-cli* tool helps in SCG data analysis, establishing a solid foundation for future research endeavors in code dependency analysis, software monitoring, maintenance, and tool development.
6. **Cost reduction in project maintenance:** With its emphasis on practical utility, the proposed SCG model and tools have the potential to influence development of future tools that help reducing project maintenance costs by expediting activities related to software comprehension.

In conclusion, the work on the Semantic Code Graph offers a rich source of inspiration and practical insights for potential researchers and experts. It encourages a shift toward empirical validation, visualization, tool development, and collaborative efforts in the realm of code dependency analysis and software comprehension, ultimately contributing to more efficient and effective software maintenance practices.

Chapter 10

Conclusions

We introduced the Semantic Code Graph (SCG), a comprehensive source code information model. The SCG is a form of software network which focuses on highly detailed source code-level dependencies, capable of representing all code definitions, declarations, and the relationships between them. SCG introduces diverse properties crucial for subsequent analysis, as well as a simple yet effective intermediate representation in the form of a protobuf schema. We have described the implementation of the SCG model and the extraction of SCG data for Scala and Java languages.

The SCG model properties enable advanced analysis and visualization of source code structure, as well as integration with external tools. Practical capabilities of SCG have been proven in empirical evaluation on eleven popular open source projects written in Java and Scala. For each project we have extracted their SCG, Class Collaboration Network, and Call Graph models, and conducted diverse comparative software comprehension experiments. Thanks to the level of detail in SCG, even a simple distribution analysis of SCG node types can provide valuable insights, such as identifying excessive usage of mutable variables and suggesting project refactoring actions.

Class Collaboration Network (CCN) and Call Graph (CG) also provide interesting perspectives on the software. CCN focuses on the high-level code structure, while CG analyzes important dependencies from a program execution perspective. Both CCN and CG can be effectively extracted from the more general SCG model. SCG, CCN, and CG exhibit a power-law distribution in each project, indicating that the software structure displays characteristics of a scale-free network, with a significant number of outlier nodes. These outlier nodes can sometimes be identified as places for direct refactoring improvements to enhance project readability and structural quality. Software partitioning of SCG, CCN, and CG models, together with graph visualization, provides valuable insight into the software architecture from the structural and operational perspective and has the potential to aid the migration process towards a more modular or even microservice-based architecture. The most important nodes in each network may be of interest for a different reasons: in the CCN network they are crucial from a project structure perspective, while in the CG graph the represent code entities having a major impact on program execution. The SCG combines these perspectives to identify a unique set of nodes critical for overall project comprehension.

The SCG model has enabled effective interactive code dependency visualization, proving its usefulness in the publicly available *Graph Buddy* plugin for popular IDEs. We have also demonstrated the utility of

SCG in general software mining (discovery of similar code snippets) leveraging data analysis in external tools, such as Jupyter Notebook with Python data frames environment.

We argue that research presented in this dissertation, which focuses on empirical experiments on real projects, helps address the gap between theoretical code representation models and their applications, as well as the surprising lack of concrete and widely used software comprehension tools in the common developer workflow. To encourage future research in this direction, i.e., one that emphasizes practical usability aspects of the underlying theoretical work, we have published our tools and focused on their usability, notably *scg-cli*, a tool for SCG extraction¹ and analysis. We hope that our work will contribute to the emergence of practical research and applications in the code structure analysis and software comprehension field.

¹Currently *scg-cli* supports SCG extraction for Java projects. For Scala, which requires a different approach, we have developed a compiler plugin for this purpose available at <https://github.com/VirtusLab/scg-scala>

Bibliography

- [1] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd international conference on software engineering*, pages 746–755, 2011.
- [2] Nedhal A. Al-Saiyd. Source code comprehension analysis in software maintenance. In *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*, pages 1–5, July 2017.
- [3] Rakan Alanazi, Gharib Gharibi, and Yugyung Lee. Facilitating program comprehension with call graph multilevel hierarchical abstractions. *Journal of Systems and Software*, 176:110945, 2021.
- [4] Rakan Alanazi, Gharib Gharibi, and Yugyung Lee. Facilitating program comprehension with call graph multilevel hierarchical abstractions. *Journal of Systems and Software*, 176:110945, 2021.
- [5] Karim Ali, Xiaoni Lai, Zhaoyi Luo, Ondřej Lhoták, Julian Dolby, and Frank Tip. A study of call graph construction for jvm-hosted languages. *IEEE Transactions on Software Engineering*, 47(12):2644–2666, Dec 2021.
- [6] Karim Ali, Marianna Rapoport, Ondřej Lhoták, and Frank Tip. Constructing call graphs of scala programs. 07 2014.
- [7] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.
- [8] Ritu Arora and Sanjay Goel. Javarelationshipgraphs (jrg): Transforming java projects into graphs using neo4j graph databases. In *Proceedings of the 2nd International Conference on Software Engineering and Information Management, ICSIM 2019*, page 80–84, New York, NY, USA, 2019. Association for Computing Machinery.
- [9] Ritu Arora, Sanjay Goel, and R. K. Mittal. Using dependency graphs to support collaboration over github: The neo4j graph database approach. In *2016 Ninth International Conference on Contemporary Computing (IC3)*, pages 1–7, Aug 2016.
- [10] Vinay Arora, Rajesh Kumar Bhatia, and Maninder Pal Singh. Evaluation of flow graph and dependence graphs for program representation. *International Journal of Computer Applications*, 56:18–23, 2012.

- [11] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 334–349, April 2017.
- [12] A. Bandi, B. J. Williams, and E. B. Allen. Empirical evidence of code decay: A systematic mapping study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 341–350, 2013.
- [13] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Using structural and semantic measures to improve software modularization. *Empirical Software Engineering*, 18(5):901–932, 2013.
- [14] Laure Bedu, Olivier Tinh, and Fabio Petrillo. A tertiary systematic literature review on software visualization. In *2019 Working Conference on Software Visualization (VISOFT)*, pages 33–44, 2019.
- [15] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 419–429, June 2012.
- [16] Roberto Almeida Bittencourt and Dalton Dario Serey Guerrero. Comparison of graph clustering algorithms for recovering software architecture module views. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 251–254, March 2009.
- [17] Johannes Bohnet and Jürgen Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *Proceedings of the 2006 ACM Symposium on Software Visualization, SoftVis '06*, pages 95–104, New York, NY, USA, 2006. Association for Computing Machinery.
- [18] Phillip Bonacich. Power and centrality: A family of measures. *American journal of sociology*, 92(5):1170–1182, 1987.
- [19] Krzysztof Borowski, Bartosz Baliś, and Tomasz Orzechowski. Graph buddy — an interactive code dependency browsing and visualization tool. In *2022 Working Conference on Software Visualization (VISOFT)*, 2022.
- [20] Krzysztof Borowski and Bartosz Baliś. scg-cli – a tool supporting software comprehension via extraction and analysis of semantic code graph, 2023. arXiv:2310.03044 [cs.SE].
- [21] Krzysztof Borowski, Bartosz Baliś, and Tomasz Orzechowski. Semantic code graph – an information model to facilitate software comprehension, 2023. arXiv:2310.02128 [cs.SE].
- [22] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25, 03 2004.
- [23] Alain Bretto. Hypergraph theory. *An introduction. Mathematical Engineering. Cham: Springer*, 1, 2013.
- [24] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998.

- [25] Aline Brito, Andre Hora, and Marco Tulio Valente. Refactoring graphs: Assessing refactoring over time. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 367–377, Feb 2020.
- [26] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent advances in graph partitioning*. Springer, 2016.
- [27] Umit V Catalyürek and Cevdet Aykanat. Patoh: A multilevel hypergraph partitioning tool, version 3.0. *Bilkent University, Department of Computer Engineering, Ankara*, 6533:4–7, 1999.
- [28] Ümit V Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs). In *Encyclopedia of parallel computing*, pages 1479–1487. Springer, 2011.
- [29] Máté Cserép and Anett Fekete. Integration of incremental build systems into software comprehension tools. In *ICAI*, pages 85–93, 2020.
- [30] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.
- [31] Xin Du, Tian Wang, Weifeng Pan, Muchou Wang, Bo Jiang, Yiming Xiang, Chunlai Chai, Jiale Wang, and Chengxiang Yuan. Cospa: Identifying key classes in object-oriented software using preference aggregation. *IEEE Access*, 9:114767–114780, 2021.
- [32] B. Du Bois, S. Demeyer, and J. Verelst. Refactoring - improving coupling and cohesion of existing code. In *11th Working Conference on Reverse Engineering*, pages 144–151, Nov 2004.
- [33] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, André C. Hora, Jannik Laval, and Tudor Gîrba. Mse and famix 3.0: an interexchange format and source code model family. 2011.
- [34] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, Jan 2001.
- [35] Ersin Ersoy, Kamer Kaya, Metin Altınişik, and Hasan Sözer. Using hypergraph clustering for software architecture reconstruction of data-tier software. In Bedir Tekinerdogan, Uwe Zdun, and Ali Babar, editors, *Software Architecture*, pages 326–333, Cham, 2016. Springer International Publishing.
- [36] Sinan Eski and Feza Buzluca. An automatic extraction approach: Transition to microservices architecture from monolithic application. In *Proceedings of the 19th International Conference on Agile Software Development: Companion, XP ’18*, New York, NY, USA, 2018. Association for Computing Machinery.
- [37] Daniel Falci, Orlando Gomes, and Fernando Silva Parreiras. Complex networks analysis for software architecture: an hibernate call graph study. 06 2017.

- [38] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.
- [39] Martin Fowler. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.
- [40] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.
- [41] Carlos Galindo, Sergio Pérez, and Josep Silva. Data dependencies in object-oriented programs. In *11th Workshop on Tools for Automatic Program Analysis*, 2020.
- [42] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 486–496. IEEE, 2013.
- [43] Sergi Gomez. *Centrality in Networks: Finding the Most Important Nodes*, pages 401–433. Springer International Publishing, Cham, 2019.
- [44] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, oct 1997.
- [45] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen. *Modern Compiler Design*, pages 1–51. Springer New York, New York, NY, 2012.
- [46] Yang Guo, Zheng-xu Zhao, and Wei Wang. Complexity analysis of software based on function-call graph. In Ershi Qi, Jiang Shen, and Runliang Dou, editors, *The 19th International Conference on Industrial Engineering and Engineering Management*, pages 269–277, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [47] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, 1988.
- [48] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Code-searchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.
- [49] David Hyland-Wood, David Carrington, and Simon Kaplan. Scale-free nature of java software package, class and method collaboration graphs. In *Proceedings of the 5th International Symposium on Empirical Software Engineering*. Citeseer, 2006.
- [50] Ayaz Isazadeh, Habib Izadkhah, and Islam Elgedawy. *Techniques for the Evaluation of Software Modularizations*, pages 179–216. Springer International Publishing, Cham, 2017.

- [51] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PloS one*, 9(6):e98679, 2014.
- [52] S. Jenkins and S.R. Kirk. Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. *Information Sciences*, 177(12):2587–2601, 2007.
- [53] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. Challenges when moving from monolith to microservice architecture. In Irene Garrigós and Manuel Wimmer, editors, *Current Trends in Web Engineering*, pages 32–47, Cham, 2018. Springer International Publishing.
- [54] Sesha Kalyur and G.S. Nagaraja. Paracite: Auto-parallelization of a sequential program using the program dependence graph. In *2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, pages 7–12, Oct 2016.
- [55] Stephen H Kan. *Metrics and models in software quality engineering*. Addison-Wesley Professional, 2003.
- [56] Abdul vahab karuthedath, Sreekutty Vijayan, and Vipin Kumar K. S. System dependence graph based test case generation for object oriented programs. In *2020 International Conference on Power, Instrumentation, Control and Computing (PICC)*, pages 1–6, Dec 2020.
- [57] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.
- [58] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.
- [59] Jinhyun Kim, HyukGeun Choi, Hansang Yun, and Byung-Ro Moon. Measuring source code similarity by finding similar subgraph with an incremental genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16*, pages 925–932, New York, NY, USA, 2016. Association for Computing Machinery.
- [60] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in Computer Virology*, 7(4):233–245, 2011.
- [61] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309, Oct 2001.
- [62] Dilshodbek Kuryazov, Dilshod Jabborov, and Bekmurod Khujamuratov. Towards decomposing monolithic applications into microservices. In *2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT)*, pages 1–4, Oct 2020.
- [63] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, volume 15, 2011.

- [64] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 185–194, May 2010.
- [65] Thomas D. LaToza and Brad A. Myers. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 117–124, 2011.
- [66] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution, IWPSE-EVOL '11*, pages 41–50, New York, NY, USA, 2011. Association for Computing Machinery.
- [67] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, 2013.
- [68] Hao Li, Tian Wang, Weifeng Pan, Muchou Wang, Chunlai Chai, Pengyu Chen, Jiale Wang, and Jing Wang. Mining key classes in java projects by examining a very small number of classes: A complex network-based approach. *IEEE Access*, 9:28076–28088, 2021.
- [69] Huan Liu, Yubo Tao, Wenda Huang, and Hai Lin. Visual exploration of dependency graph in source code via embedding-based similarity. *Journal of Visualization*, 24(3):565–581, 2021.
- [70] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidović, and Robert Kroeger. Measuring the impact of code dependencies on software architecture recovery techniques. *IEEE Transactions on Software Engineering*, 44(2):159–181, Feb 2018.
- [71] Anna-Liisa Mattila, Petri Ihanola, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Väättäjä. Software visualization today: Systematic literature review. In *Proceedings of the 20th International Academic Mindtrek Conference, AcademicMindtrek '16*, pages 262–271, New York, NY, USA, 2016. Association for Computing Machinery.
- [72] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.
- [73] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.
- [74] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. Modeling functional similarity in source code with graph-based siamese networks. *IEEE Transactions on Software Engineering*, 48(10):3771–3789, Oct 2022.
- [75] Yulong Meng, Dong Xu, Ziyang Zhang, and Wencai Li. System dependency graph construction algorithm based on equivalent substitution. In *2015 Eighth International Conference on Internet Computing for Science and Engineering (ICICSE)*, pages 106–110, 2015.

- [76] P. Meyer, Harvey Siy, and Sanjukta Bhowmick. Identifying important classes of large software systems through k-core decomposition. *Advances in Complex Systems*, 17:1550004, 04 2015.
- [77] Justin Middleton and Kathryn T. Stolee. Understanding similar code through comparative comprehension. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–11, Sep. 2022.
- [78] Sina Mohammadi and Habib Izadkhah. A new algorithm for software clustering considering the knowledge of dependency between artifacts in the source code. *Information and Software Technology*, 105:252–256, 2019.
- [79] Christopher R Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical review E*, 68(4):046116, 2003.
- [80] Aravind Nair, Avijit Roy, and Karl Meinke. Funcgnn: A graph neural network approach to program similarity. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.
- [81] M. Newman. *Networks*. OUP Oxford, 2018.
- [82] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.
- [83] Martin Odersky, Eugene Burmako, and Dmytro Petrashko. Tasty reference manual. page 21, 2016.
- [84] Rocco Oliveto, Malcom Gethers, Gabriele Bavota, Denys Poshyvanyk, and Andrea De Lucia. Identifying method friendships to remove the feature envy bad smell: Nier track. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 820–823, May 2011.
- [85] Michael P O'brien. Software comprehension—a review & research direction. *Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report*, 2003.
- [86] Weifeng Pan, Beibei Song, Kangshun Li, and Kejun Zhang. Identifying key classes in object-oriented software using generalized k-core decomposition. *Future Generation Computer Systems*, 81:188–202, 2018.
- [87] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.
- [88] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7, Nov 2019.

- [89] Zoltán Porkoláb, Tibor Brunner, Dániel Krupp, and Márton Csordás. Codecompass: An open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 361–369, New York, NY, USA, 2018. Association for Computing Machinery.
- [90] Zoltán Porkoláb, Tibor Brunner, Dániel Krupp, and Márton Csordás. Codecompass: an open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension*, pages 361–369, 2018.
- [91] Babak Poursaghar, Habib Izadkhah, Ayaz Isazadeh, and Shahriar Lotfi. A graph-based clustering algorithm for software systems modularization. *Information and Software Technology*, 133:106469, 2021.
- [92] Christian R. Prause and Stefan Apelt. An approach for continuous inspection of source code. In *Proceedings of the 6th International Workshop on Software Quality*, WoSQ '08, pages 17–22, New York, NY, USA, 2008. Association for Computing Machinery.
- [93] Mehwish Riaz, Muhammad Sulayman, and Husnain Naqvi. Architectural decay during continuous software evolution and impact of ‘design for change’ on software architecture. In Dominik Ślęzak, Tai-hoon Kim, Akingbehin Kiumi, Tao Jiang, June Verner, and Silvia Abrahão, editors, *Advances in Software Engineering*, pages 119–126, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [94] Yannick Rochat. Closeness centrality extended to unconnected graphs: The harmonic centrality index. Technical report, 2009.
- [95] Oscar Rodriguez-Prieto, Alan Mycroft, and Francisco Ortin. An efficient and scalable platform for java source code analysis using overlaid graph representations. *IEEE Access*, 8:72239–72260, 2020.
- [96] Linda H. Rosenberg and Lawrence E. Hyatt. Software quality metrics for object-oriented system environments. 1995.
- [97] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.
- [98] B.G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.
- [99] Qusay I. Sarhan, Bestoun S. Ahmed, Miroslav Bures, and Kamal Z. Zamli. Software module clustering: An in-depth literature analysis. *IEEE Transactions on Software Engineering*, 48(6):1905–1928, June 2022.
- [100] Miloš Savić, Miloš Radovanović, and Mirjana Ivanović. Community detection and analysis of community evolution in apache ant class collaboration networks. In *Proceedings of the Fifth Balkan Conference in Informatics*, pages 229–234, 2012.

- [101] Miloš Savić, Gordana Rakić, Zoran Budimac, and Mirjana Ivanović. A language-independent approach to the extraction of dependencies between source code entities. *Information and Software Technology*, 56(10):1268–1288, 2014.
- [102] Anthony Savidis. and Crystallia Savaki. Software architecture mining from source code with dependency graph clustering and visualization. In *Proceedings of the 17th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2022) - IVAPP*, pages 179–186. INSTICC, SciTePress, 2022.
- [103] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. Comprehending studies on program comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 308–311, May 2017.
- [104] Michael D. Shah and Samuel Z. Guyer. An interactive microarray call-graph visualization. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 86–90, Oct 2016.
- [105] G. Shu, B. Sun, T. A. D. Henderson, and A. Podgurski. JavaPDG: A New Platform for Program Dependence Analysis. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 408–415, 2013.
- [106] Gang Shu, Boya Sun, Tim A.D. Henderson, and Andy Podgurski. JavaPDG: A new platform for program dependence analysis. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, mar 2013.
- [107] Janet Siegmund. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 13–20, March 2016.
- [108] Michael Smit, Barry Gergel, H. James Hoover, and Eleni Stroulia. Code convention adherence in evolving software. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 504–507, Sep. 2011.
- [109] Nicholas Smith, Danny Van Bruggen, and Federico Tomassetti. Javaparser: visited. *Leanpub*, oct. de, 10:29–40, 2017.
- [110] Muttipati Srinivasu and Poosapati Padmaja. Unfolding and boosting based graph partitioning approach for object-oriented system. *IARJSET*, 3:122–128, 07 2016.
- [111] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension, ICPC ’20*, pages 2–13, New York, NY, USA, 2020. Association for Computing Machinery.
- [112] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Laredo, and Alessandro Morari. Learning to map source code to software vulnerability using code-as-a-graph. *ArXiv*, abs/2006.08614, 2020.

- [113] Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.
- [114] Divya Tyagi, Ritu Arora, and Yashvardhan Sharma. Application of java relationship graphs to academics for detection of plagiarism in java projects. In Milan Tuba, Shyam Akashe, and Amit Joshi, editors, *ICT Systems and Sustainability*, pages 761–772, Singapore, 2022. Springer Nature Singapore.
- [115] Raoul-Gabriel Urma and Alan Mycroft. Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming*, 97:127–134, 2015.
- [116] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.
- [117] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. Tracking design smells: Lessons from a study of god classes. In *2009 16th working conference on reverse engineering*, pages 145–154. IEEE, 2009.
- [118] N. Walkinshaw, M. Roper, and M. Wood. The java system dependence graph. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 55–64, Sep. 2003.
- [119] R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 45–54, Sep. 2003.
- [120] Hadley Wickham. The split-apply-combine strategy for data analysis. *Journal of statistical software*, 40:1–29, 2011.
- [121] Spiros Xanthos and North Goodwin. Clustering object-oriented software systems using spectral graph partitioning. *Urbana*, 51(1):1–5, 2006.
- [122] Hong Xia, Yongkang Zhang, Yanping Chen, Hengshan Zhang, Zhongmin Wang, and Fengwei Wang. Software module clustering using the hierarchical clustering combination method. In *2022 7th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, pages 155–160, April 2022.
- [123] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, Oct 2018.
- [124] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.
- [125] Siham Yousfi and Dalila Chiadmi. Graph file format for etl. In *2014 Second World Conference on Complex Systems (WCCS)*, pages 189–195, Nov 2014.

- [126] Dongjin Yu, Quanxin Yang, Xin Chen, Jie Chen, and Yihang Xu. Graph-based code semantics learning for efficient semantic code clone detection. *Information and Software Technology*, 156:107130, 2023.
- [127] Jianjun Zhao. Applying program dependence analysis to java software. 06 2001.