International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347-6435(Online) Volume 11, Issue 11, November 2022)

A Review and Analysis of Java Source Code using Graph Representation

Shubham Kumar[1], Prof. Sarwesh Site[2]

[1]M.Tech Scholar, [2]Assistant Professor, Department of Computer Science and Engineering, All Saints' College of Technology, Bhopal, India

***Abstract*--This information is used for many different purposes, such as static program analysis, advanced code search, coding guideline checking, software metrics computation, and extraction of semantic and syntactic information to create predictive models. Most of the existing systems that provide these kinds of services are designed ad hoc for the particular purpose they are aimed at. For this reason, we created ProgQuery, a platform to allow users to write their own Java program analyses in a declarative fashion, using graph representations. We modify the Java compiler to compute seven syntactic and semantic representations, and store them in a Neo4j graph database. Such representations are overlaid, meaning that syntactic and semantic nodes of the different graphs are interconnected to allow combining different kinds of information in the queries/analyses. We evaluate ProgQuery and compare it to the related systems.**

I. Introduction

JAVA was developed by James Gosling at Sun Microsystems Inc. in the year 1995, later acquired by Oracle Corporation. It is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs. Java is a class-based, object-oriented programming language and is designed to have as few implementation dependencies as possible. A general-purpose programming language made for developers to write once run anywhere that is compiled Java code can run on all platforms that support Java. Java applications are compiled to byte code that can run on any Java Virtual Machine. The syntax of Java is similar to c/c++.

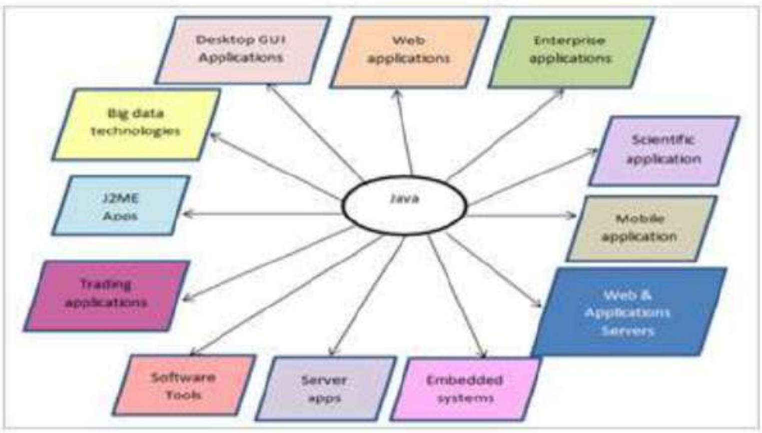

Figure 1: Use of Java In Different Technology.

History: Java's history is very interesting. It is a programming language created in 1991. James Gosling, Mike Sheridan, and Patrick Naughton, a team of Sun engineers known as the Green team initiated the Java language in 1991. Sun Microsystems released its first public implementation in 1996 as Java 1.0.

It provides no-cost -run-times on popular platforms. Java1.0 compiler was re-written in Java by Arthur Van Hoff to strictly comply with its specifications. With the arrival of Java 2, newversions had multiple configurations built for different types of platforms.

II. Methodology

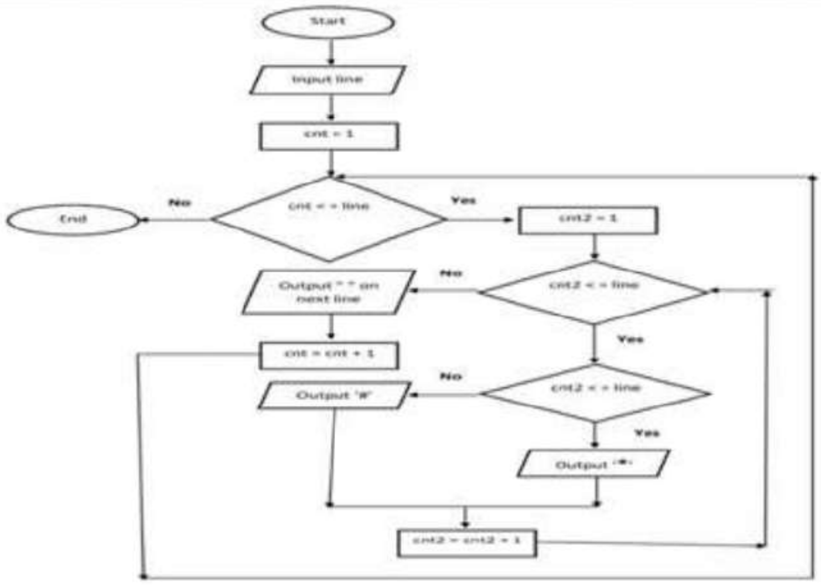

Figure 2: Flow Chart of Java

The work presented in this article is inspired by Wiggle, a prototype source-code querying system based on the graph data model [9]. Wiggle modifies the Java compiler to obtain ASTs of each program and store them in a Neo4j graph database. These persistent ASTs may be consulted using any of the mechanisms provided by Neo4j, such as the Cypher query language. Wiggle uses overlays as a mechanism to express queries as a mixture of syntactic and semantic information [20]. The prototype implementation provides limited semantic data about type hierarchy and attribution (annotation), and method calls.

Frappé is a C/C++ source code query tool that supports large-scale codebases [25]. A Neo4j graph database is chosen to gain query efficiency by avoiding repeated join operations, necessary in the relational model. Frappé provides a modification of the Clang compiler to retrieve and store program information. It provides some scripts that execute the Clang modification (to insert program information) and the original C compiler (to generate binary code). Although Frappé stores some AST information, important nodes such as expressions and statements are not included in the representation. It does not include such nodes to provide good performance for large codebases. Frappé represents node types with a string property, but it does not support different types (subtyping polymorphism). A call dependency graph is provided, connecting function nodes through *calls* relationships.

^{1}Program representations are stored in Mango DB, a Python wrapper for MongoDB [29]. JSON documents in the database represent syntactic and semantic information of language-agnostic programs. The semantic representations include type hierarchy, data dependency and method call graphs. No information about control flow or type dependency is stored. For source code queries, they propose JIns$^+$, an extension of their JIns declarative code instrumentation language[30]. Users may use JIns$^+$ to write their own analyses, valid for any object-oriented language. Although expressions are stored in the database, JIns$^+$ does not allow queries about expressions. Therefore, common queries such as locating expressions calling a method or using a variable cannot be expressed. This framework reports analyses results as JSON documents.

Its own additional analyses and metrics. SonarQube finds not only bugs but also bad smells, which do not prevent correct program functioning, but usually correspond to another problem in the system [33] (e.g., code duplication, forgotten interfaces and orphanabstract classes). SonarQube can be extended by implementing new Java user-defined plug-ins, consisting of one or more analyzers. Its GUI allows hierarchical inspection of source code and provides multiple views and code statistics (e.g., unit-test coverage, code duplications, and documentation and coupling metrics).

III. Deployment Result

1) MET53-J (program understandability) [50]. Ensure that the clone method calls super.clone(). clone may call another method that transitively calls super.clone().
2) MET55-J (reliability) [36]. Return an empty array or collection instead of a null value for methods that return an array or collection. We check all the return statements and all the types implementing the Collection interface.
3) SEC56-J (reliability) [48]. Do not serialize direct handles to system resources. Serialized objects can be altered outside ofany Java program, implying potential vulnerabilities. We detect types implementing Serializable with non-transient fields derived from system resources such as File, Naming Context and Domain Manager.
4) DCL56-J (defensive programming) [36], [48], [50]. Do not attach significance to the ordinal associated with an enum. If the ordinal method is invoked, this analysis encourages the programmer to replace it with an integer field.
5) MET50-J (program understandability) [36], [50]. Avoid ambiguous or confusing uses of overloading. This analysis detects classes with overloaded methods with a) the same parameter types in a different order; or b) four or more parameters in different implementations.

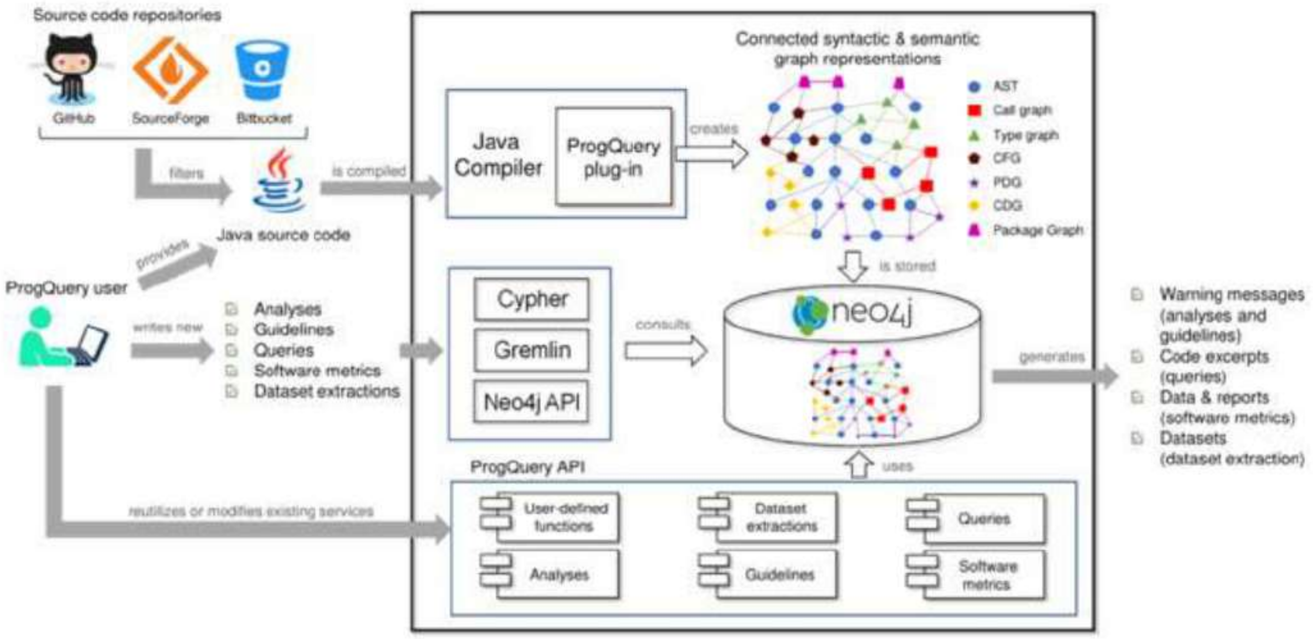

Figure 3: Java Analysis Source Code

It effectively and efficiently detects common defects that developers will want to review and correct. It was designed to avoid generating false warnings (false positives). FindBugs implements control-flow and intra-procedural dataflow analyses. It follows a plug-in architecture that allows users to write their own analyses (detectors) in Java. Detectors are commonly implemented with the *Visitor* design pattern [31].

Detectors may traverse the AST, type hierarchies and control- and data-flow graphs. Users can runFindBugs from the command line, and it provides plug-ins for Eclipse, NetBeans, Ant and Maven. It also implements a GUI that supports the inspection of analysis results. Find Bugs does not store program information in a database. Analysis results are savedas XML documents.

IJRDET
International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347-6435(Online) Volume 11, Issue 11, November 2022)

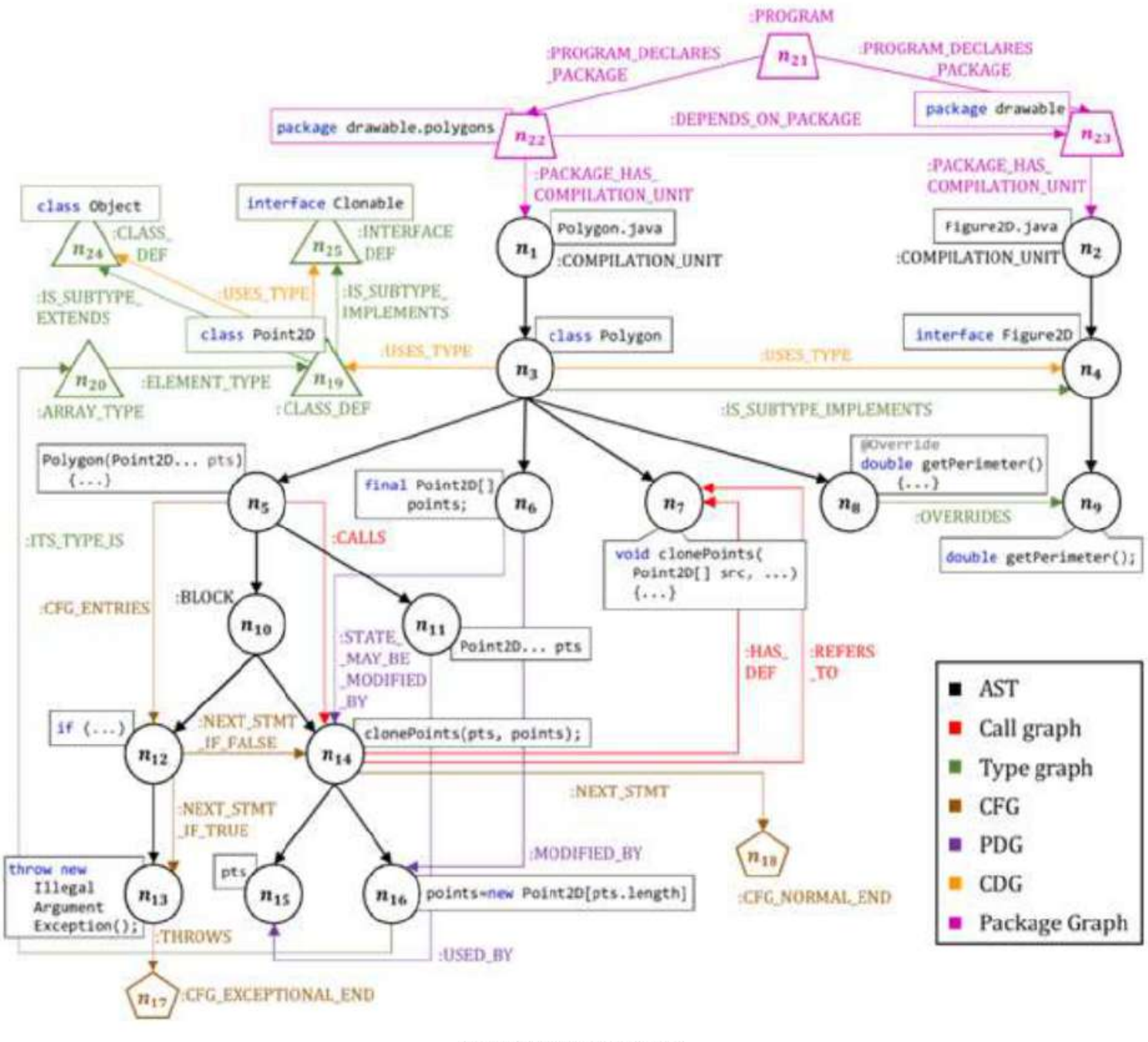

Figure 4: Java Call Graph

CFG also models the exceptional jumps performed by Java checked exceptions,[1] and assert and throw instructions. For that purpose, ProgQuery defines CFG nodes representing exceptional method termination, and exceptions handled in a catchorfinallyblock.Forexceptionhandling, thestatic types of the exceptions thrown and caught are analyzed, connecting them only if they could match at runtime—these kinds of connections represent *may* relationships, whereas NEXT_STATEMENT and CFG_ENTRIES represent *must* relationships. In Figure 4, the throw statement represented by $n13$ is connected to the CFG_EXCEPTIONAL_END node $n17$ through a *must* THROWS relationship, because no catch or finally blocks are used to handle the exception.

IV. Conclusion

1) Can the ontology defined in Appendix A be used to express real static program analyses?
2) Does ProgQuery provide reduced analysis times compared to existing approaches?
3) Does it provide better scalability for increasing program sizes?
4) Does it provide better scalability for increasing analysis complexity?
5) Is it able to perform complex analyses against huge Java programs in a reasonable time?
6) Can program analyses be expressed succinctly, in a declarative manner, and using standard query-language syntax?
7) Are there any drawbacks of our system, compared to related approaches?

REFERENCES

[1] A. W. Appel and J. Palsberg, Modern Compiler Implementation in Java, 2nd ed. New York, NY, USA: Cambridge Univ. Press, 2003.

[2] R.-G. Urma and A. Mycroft, ''Programming language evolution via source code Query languages,'' in Proc. ACM 4th Annu. Workshop Eval. UsabilityProgram. Lang. Tools, New York, NY, USA, 2012, pp. 35–38.

[3] F. Nielson, H. R. Nielson, and C. Hankin, Principles of Program Analysis.Cham, Switzerland: Springer, 2010.

[4] F. Ortin, J. Escalada, and O. Rodriguez-Prieto, ''Big code: New opportunities for improving software construction,'' J. Softw., vo0l. 11, no.11, pp. 1083–1088, 2016.

[5] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, ''Matching dependence-related queries in the system dependence graph,'' in Proc. IEEE/ACM Int. Conf. Automated Softw. Eng., New York, NY, USA, 2010,pp. 457–466.

[6] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, ''Using static analysis to find bugs,'' IEEE Softw., vol. 25, no. 5, pp. 22– 29, Sep. 2008.

[7] R. Ivanov. (2020). Checkstyle. [Online]. Available: https://checkstyle. sourceforge.io

[8] Coverity. (2020). Coverity Scan Static Analysis. [Online]. Available: https://scan.coverity.com

[9] R.-G. Urma and A. Mycroft, ''Source-code queries with graph databases— With application to programming language usage and evolution,'' Sci. Comput. Program., vol. 97, pp. 127–134, Jan. 2015.

[10] Google. (2020). Big Query. [Online]. Available: https://cloud. google.com/bigquery

[11] R.-G. Urma. (2020). Wiggle. [Online]. Available: https://github. com/raoulDoc/WiggleIndexer

[12] V. Raychev, M. Vechev, and A. Krause, ''Predicting program properties from 'Big Code,''' in Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang., New York, NY, USA, 2015, pp. 111–124.

[13] Defense Advanced Research Projects Agency. (2014). MUSE Envisions Mining, 'Big Code' to Improve Software Reliability and Construction. [Online]. Available: http://www.darpa.mil/news-events/2014-03-06a,

[14] S. Karaivanov, V. Raychev, and M. Vechev, ''Phrase-based statistical translation of programming languages,'' in Proc. ACM Int. Symp. New Ideas, New Paradigms, Reflections Program. Softw., New York, NY, USA,2014, pp. 173–184.

[15] F. Yamaguchi, M. Lottmann, and K. Rieck, ''Generalized vulnerability extrapolation using abstract syntax trees,'' in Proc. 28th Annu. Comput. Secur. Appl. Conf., New York, NY, USA, 2012, pp. 359–368.

[1] In Java, unchecked exceptions are Runtime Exception, Error and their subclasses.