IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 25, 2020

Digital Object Identifier 10.1109/ACCESS.2020.DOI

An Efficient and Scalable Platform for Java Source Code Analysis using Overlaid Graph Representations

OSCAR RODRIGUEZ-PRIETO[1], ALAN MYCROFT[2], and FRANCISCO ORTIN[1,3]
[1]Department of Computer Science, University of Oviedo, c/Federico Garcia Lorca 18, 33007, Oviedo, Spain
[2]Department of Computer Science and Technology, University of Cambridge, UK
[3]Department of Computer Science, Cork Institute of Technology, Rossa Ave., Bishopstown, Cork, T12 P928, Ireland

Corresponding author: Francisco Ortin (e-mail: ortin@uniovi.es).

This work has been partially funded by the Spanish Department of Science, Innovation and Universities: project RTI2018-099235-B-I00. The first and third authors have also received funds from the University of Oviedo through its support to official research groups (GR-2011-0040).

ABSTRACT Although source code programs are commonly written as textual information, they enclose syntactic and semantic information that is usually represented as graphs. 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. Our platform outperforms the other systems in analysis time, and scales better to program sizes and analysis complexity. Moreover, the queries coded show that ProgQuery is more expressive than the other approaches. The additional information stored by ProgQuery increases the database size and associated insertion time, but these increases are significantly lower than the query/analysis performance gains obtained.

INDEX TERMS Code analysis, graph database, coding guidelines, declarative query language, program representation, Cypher, Java, Neo4j.

I. INTRODUCTION

In textual programming languages, input programs are collections of source code files (plus additional resources) coded as text. That textual information actually encloses syntactic and semantic information that compilers and interpreters, after different analysis phases, represent internally as tree and graph structures [1]. Besides compilers, other tools use that syntactic and semantic information for different purposes such as advanced source code querying [2], program analysis [3], and constructing predictive tools that learn from massive source code repositories [4].

Software developers and tools often query source code to explore a system, or to search for code for maintenance tasks such as bug fixing, refactoring and optimization. Text-based search techniques are mainly based on finding sequences of words and regular expressions, but they do not support advanced queries based on syntactic and semantic properties of the source code. That is a common necessity when, for instance, a programmer wants to locate all the occurrences of a code pattern in order to refactor them [5].

Program analysis is another case scenario where syntactic and semantic information of source code is used for different purposes. Static program analysis is based on the evaluation of different dynamic program properties—such as correctness, optimization, robustness and safety—without executing the source code [3]. For example, FindBugs is an open-source static-analysis tool that finds potential bugs and performance issues in Java code, not detected by the compiler (e.g., some

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

`null` pointer dereferences) [6]. Checkstyle checks whether Java source code conforms to specific coding conventions and standards (e.g., Sun code conventions and Google Java style), widely used to automate code review processes [7]. Coverity is a multi-language commercial product that identifies critical software defects and security vulnerabilities (e.g., buffer overflow) [8]. These tools work with different syntactic and semantic representations of source code, but users cannot specify their own ad-hoc analyses/queries via a standard language. Instead, they have to implement their own code analyzer by calling the APIs or services provided by the tools.

In the existing advanced program analysis and querying tools, there is a trade-off between scalability and the source-code detail provided [9]. Those that scale to million-line programs use relational or deductive databases, but they just provide a reduced subset of all the program information, with consequent limitations on their query expressiveness. For example, Google's BigQuery provides a scalable SQL search system for any project in GitHub [10], but syntactic and semantic patterns cannot be consulted. On the other hand, some tools provide more advanced program information, but that affects the scalability of the system. For example, Wiggle provides detailed syntactic structures of Java programs [11], but some non-basic analyses do not terminate when applied to 60K lines of code programs (Section V-B3).

As mentioned, syntactic and semantic information of source code is also used in the construction of predictive tools that learn from massive code repositories (e.g., GitHub, SourceForge, BitBucket and CodePlex) using machine learning and probabilistic reasoning [12]. The information extracted from programs in code repositories represents a huge amount of data to build predictive models. This research field is commonly referred to as "big code", since it brings together big data and code analysis [13]. The big code approach has been used to build tools such as deobfuscators [12], statistical machine translation [14], security vulnerability detection [15] and decompilation systems [16]. These tools extract syntactic and semantic information from programs by implementing ad-hoc extraction procedures, depending on the necessities of each particular problem. However, they do not allow the user to easily extract syntactic and semantic information from programs in a standardized declarative fashion.

The storage of program representations is an important issue to be considered when building a platform for source code analysis. The persistence system must allow the storage of multiple graph representations of million-line programs. Information must be provided efficiently, because static program analyses consult much information in different program representations.

Graph databases support the native storage of graph structures, represented with nodes, relationships (edges) and properties (data stored for nodes and relationships). On the other hand, relational databases represent entities as tuples (set of values), grouped into tables. Relationships between entities are represented with common attributes in different tables. In graph databases, records contain direct pointers to connected nodes. This feature avoids the need to link tables by checking the values of common attributes, as occurs in relational databases. Therefore, the storage of graph structures in relational databases causes an impedance mismatch that may involve performance costs [17]. Another benefit of graph databases is that they provide declarative graph query languages to retrieve and update information from the persistent store, maintaining graph abstractions.

The main contribution of this article is an efficient and scalable platform, called ProgQuery, that allows the user to perform advanced program analyses, queries and feature extractions, expressed by means of advanced syntactic and semantic program properties, in a declarative and expressive way. Our work is based on the idea that programs can be represented with graph structures that connect different syntactic and semantic information, stored in a graph database [9]. Using our open-source system, users can write their own queries and analyses using an expressive declarative query language. The system is able to perform in seconds complex analyses against huge Java programs with tens of millions of lines of code.

The rest of the paper is structured as follows. The next section describes a motivating example, and related work is discussed in Section III. Section IV describes our platform. We evaluate ProgQuery in Section V, and compare it with the related systems. Section VI presents the conclusions and future work.

II. MOTIVATING EXAMPLE

In order to illustrate how our system works, we analyze the OBJ50-J Java recommendation published by the CERT division of the Software Engineering Institute at Carnegie Mellon University [18]. That recommendation is titled "*never confuse the immutability of a reference with that of the referenced object*". In Java, the value of `final` references cannot be modified (e.g., line 10 in Figure 1), meaning that it is not possible to change which object they point to. However, a common incorrect assumption is that `final` prevents modification of the state of the object pointed to by the reference (i.e., believing the compiler prompts an error in line 26 of Figure 1). Therefore, many programmers improperly use `final` references, when they actually want avoid mutability of the referenced object.

Figure 2 shows how we implemented the OBJ50-J analysis in our system. As mentioned, ProgQuery models programs with different graph structures representing different syntactic and semantic information. Those representations are stored in a Neo4j graph database, which provides the Cypher declarative graph query language [19]. The code in Figure 2 is a Cypher query that uses graph-based program representations to implement the OBJ50-J analysis. It shows an informative warning message to the user when the recommendation is not fulfilled.

The Cypher code in line 01 (Figure 2) matches all

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access


```
01: package drawable;
02:
03: public interface Figure2D {
04:   double getPerimeter();
05: }
```

```
06: package drawable.polygons;
07:
08: public class Polygon implements Figure2D {
09:
10:   final Point2D[] points;
11:
12:   public Polygon(Point2D... pts) {
13:     if (pts.length < 3)
14:       throw new IllegalArgumentException(
                  "A polygon must have at least three vertices.");
15:     clonePoints(pts, points = new Point2D[pts.length]);
16:   }
17:
18:   private static void clonePoints(Point2D[] src, Point2D[] dest){
19:     if (src.length != dest.length)
20:       throw new IllegalArgumentException(
                  "Point arrays must have the same length.");
21:     for (int i = 0; i < src.length; i++)
22:       dest[i] = (Point2D) src[i].clone();
23:   }
24:
25:   public void setPoint(int index, Point2D newPoint) {
26:     points[index] = newPoint;
27:   }
28:
29:   @Override
30:   public double getPerimeter() {
31:     double perimeter = 0;
32:     int nVertices = points.length;
33:     for (int i = 0; i < nVertices; i++)
34:       perimeter += points[i].distance(points[(i+1)%nVertices]);
35:     return perimeter;
36:    }
37: }
```

FIGURE 1. Example Java code.

the `final` variables (fields, parameters and local variables) which state may be modified by a given expression (`mutatorExpr`). The only variable matched is the `final points` field in line 10 of Figure 1 (there is no other `final` variable). The mutator expressions matched for that variable are the method invocation in line 15 and the assignment in line 26 (Figure 1). Notice that the invocation in line 15 may modify the state of `points` indirectly, because it calls `clonePoints`, which modifies the object referred by its second parameter.

The `with` clause in line 02 is used to apply another match to the results of the previous one. Additionally, `mutatorMethod` is set to the method or constructor where `mutatorExpr` was defined in (functionality implemented by the ProgQuery's user-defined function `getEnclMethodFromExpr`).

The second `match` in Figure 2 gets the class (`mutatorEnclClass`) and Java file (`mutatorCU`) where the `mutatorMethods` are defined. The `where` clause discards the matched subgraphs where variables are fields (`ATTR_DEF`), fields in the mutator expression belong to the implicit object[1] (`isOwnAccess`), and the mutator method is a constructor or another non-public method only called

[1]Line 26 in Figure 1 modifies the state of the `points` reference belonging to the implicit object (i.e., the object pointed by `this`), so `mutation.isOwnAccess` in line 04 of Figure 2 is true. However, if we had "`otherPoints[index] = newPoint;`", being `otherPoints` a `Point2D[]` parameter, `mutation.isOwnAccess` would be false, because `otherPoints` would not necessarily point to the implicit object (`this`).

```
01: MATCH (variable:VARIABLE_DEF {isFinal:true})
    -[mutation:STATE_MODIFIED_BY|STATE_MAY_BE_MODIFIED_BY]
    ->(mutatorExpr)
02: WITH variable, mutation, mutatorExpr, database.
    procedures.getEnclMethodFromExpr(mutatorExpr)
    as mutatorMethod
03: MATCH (mutatorMethod)<-[:DECLARES_METHOD|
    DECLARES_CONSTRUCTOR|HAS_STATIC_INIT]-(mutatorEnclClass)
    <-[:HAS_TYPE_DEF|:HAS_INNER_TYPE_DEF]
    -(mutatorCU:COMPILATION_UNIT)
04: WHERE NOT(variable:ATTR_DEF AND mutation.isOwnAccess AND
              mutatorMethod.isInitializer)
05: WITH variable, database.procedures.
    getEnclosingClass(variable) as variableEnclClass,
    REDUCE(seed='', mutationWarn IN COLLECT( ' Line ' +
    mutatorExpr.lineNumber + ', column ' + mutatorExpr.column
    +  ', file \''+ mutatorCU.fileName + '\'') |
    seed + '\n' + mutationWarn ) as mutatorsMessage
06: MATCH (variableEnclClass)<-[:HAS_TYPE_DEF|
    :HAS_INNER_TYPE_DEF]-(variableCU:COMPILATION_UNIT)
07: RETURN 'Warning [CMU-OBJ50] The state of variable \''+
    variable.name + '\' (in line ' + variable.lineNumber +
    ', file \'' + variableCU.fileName +
    '\') is mutated, but declared final. The state of \''+
    variable.name +'\' is mutated in:' + mutatorsMessage
```

FIGURE 2. Cypher code implementing the OBJ50-J CERT CMU Java recommendation.

by a constructor (`isInitializer`). The purpose of this `where` clause is to tell field initialization from field mutation. If the object state is changed in or from the constructor, it is initialization; otherwise, it is mutation. Therefore, the mutator invocation in line 15, which modifies the object state from the constructor, is discarded.

Lines 05, 06 and 07 in Figure 2 are aimed at building and returning the warning message. Line 05 uses the common `reduce` higher order function to build a single string indicating all the code locations where the `final` variable is mutated. Line 06 simply sets to `variableCU` the file where the final variable is defined, and line 07 returns the warning message. For the Java program in Figure 1, ProgQuery prompts the following message:

> *Warning [CMU-OBJ50] The state of variable 'points' (in line 10, file 'C:\..\Polygon.java') is mutated, but declared final. The state of 'points' is mutated in: Line 26, column 17, file 'C:\..\Polygon.java'.*

This motivating example shows how ProgQuery provides multiple graph representations stored in Neo4j, which can be used to perform program analyses such as OBJ50-J. Cypher declarative query language facilitates making the most of the graph representations and user-defined functions provided by ProgQuery. Moreover, our platform provides significantly better analysis time and scalability than the existing systems, with no additional memory consumption (see Section V).

III. RELATED WORK

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

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. Information about type hierarchy and method calls consists in specific edges connecting type and method definitions, which are sometimes duplicated (instead of reused) for projects with multiple files [11]. In Wiggle, call graphs do not include constructors (just methods), and method invocation nodes are not connected to the definition of the method being invoked. ASTs are annotated with types represented as strings, making it difficult to obtain the structural information of types. The implementation gathers node type information using reflection, causing a significant performance penalty (see Section V-E). Wiggle only supports Java 7, so newer elements of the language (such as lambda expressions, method references, and default methods) are not represented. Wiggle's authors propose the representation of more sophisticated overlays, such as control and data flow, to facilitate the implementation of complex code analyses [9].

Semmle CodeQL is a source-code analysis platform for Java, C#, Python, JavaScript and C/C++ [21]. Each codebase is created by a language-specific extractor implemented from existing compiler front-ends, which takes program representations and stores them into a relational database. Users are required to upload their programs from GitHub or BitBucket, using the LTGM web application [63]. For Java, the relational database stores the AST, type information, and additional metadata. AST structures are converted into tables by storing each node as an entry, and connecting them through primary/foreign keys. Types are stored in another table, representing their hierarchical relationships. Expressions are attributed with their inferred types. Other tables store the methods and constructors statically invoked by expressions, and variables bound to their definition. Information in the database is consulted with QL, an object-oriented variant of the Datalog logical query language [22]. The analyses written in QL could be run in LGTM or using plug-ins for different IDEs such as Visual Studio, Eclipse and IntelliJ. Although QL provides graph abstractions to consult program representations, its storage in a relational database causes an impedance mismatch [23], [24]. Some evaluations have shown that graph databases perform better than relational ones when traversing graph structures [17].

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. Data dependency information relates function and variable nodes. However, since expressions are not included, Frappé represents neither where a function is called, nor where a variable is written. The `isa_type` edge relates functions and variables with their types. Frappé provides no control flow analysis. Users can write their own queries in Cypher, and run them in the Frappé GUI. Frappé has been used to query the Linux kernel (11.4 million lines of code), and to manage multiple source code versions [26].

Zhang *et al.* propose a source code query framework to support syntactic and semantic code queries across different object-oriented programming languages [27]. They handle language heterogeneity by transforming source code into a unified abstract syntax format, using TXL [28]. Program representations are stored in MangoDB, 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.

FindBugs is a static analysis tool that looks for more than 300 different bugs in Java code [6]. Unlike other tools, FindBugs does not try to identify all the defects in a source program. Rather, 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 run FindBugs 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. FindBugs does not store program information in a database. Analysis results are saved as XML documents.

SonarQube is an open-source platform for continuous inspection of code quality, which performs static code analysis to detect bugs, code smells, and security vulnerabilities on more than 20 programming languages [32]. It uses existing analysis tools that derive metrics from their output, and add its own additional analyses and metrics. SonarQube finds not only bugs but also bad smells, which do not prevent

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

correct program functioning, but usually correspond to another problem in the system [33] (e.g., code duplication, forgotten interfaces and orphan abstract 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).

Coverity is a static analysis tool that finds defects and security vulnerabilities in source code written in Java, C#, Python, JavaScript and C/C++ [8]. Code is processed and stored in a database, which is consulted later to perform code analyses. Example analyses are resource leaks, dereferences of `null` pointers, memory corruptions, buffer overruns, control flow and error handling issues, and insecure data handling. For that purpose, inter-procedural data and control flow analyses are implemented. Analyses are neither sound nor complete; there may be non-reported defects (false negatives) and false defects reported (false positives) [34]. Coverity does not allow users to write their own analyses.

PMD is an extensible cross-language static code analyzer [35]. It finds common programming flaws, such as unused variables, empty `catch` blocks, unnecessary object creation and copy-pasted code. PMD stores neither program information nor analysis results in a database. Since PMD supports different languages, it provides groups of language-specific detectors. Analyses in PMD are expressed with rules. PDM provides an extensive API to write customized rules, implemented either in Java or as a self-contained XPath query. PMD builds ASTs, type representations, and control-flow and data-flow graphs. Analyses are launched from the command line, specifying the input programs, rules to be executed and output format.

IV. PROGQUERY

Figure 3 shows the architecture of ProgQuery. The Java programs to be processed can be taken from existing open source repositories (e.g., GitHub, Source Forge and Bitbucket), or provided by the user. Java code is compiled by our modification of the standard Java compiler. We developed a plug-in that, besides generating code, creates seven different graph representations for each program. These representations are overlaid, meaning that a syntactic node may be connected to other different semantic representations through semantic relationships, and vice versa (Section IV-B). Our modified Java compiler creates the seven different overlaid representations, and stores them into a Neo4j graph database.

ProgQuery users may consult the distinct graph representations for a given program in various ways. They may write static program analyses [36], check for guideline compliance [18], search for code using advanced queries [9], obtain software metrics [37], and extract datasets with syntactic and semantic information [38]. Our platform provides a collection of existing services (analyses, queries, guidelines, etc.) as part of the ProgQuery API. In this way, users may use such existing services against their Java code. The ProgQuery API also includes helpful Neo4j user-defined functions to facilitate the creation of new analyses (e.g., `getEnclMethodFromExpr` and `getEnclosingClass` in Figure 2).

Program representations stored in Neo4j may be consulted in different ways. One mechanism is Cypher, a widespread declarative graph query language, widely used with Neo4j [19]. Another approach is Gremlin, a graph traversal language that supports imperative and declarative querying for Neo4j (among other graph databases) [39]. ProgQuery users can also consult the graph representations in Neo4j by using its traversal framework API, a callback-based lazily-executed system to specify desired movements through graphs [40].

Given Java source code and a query, ProgQuery generates various types of output. For program analyses and guideline compliances, ProgQuery returns a collection of warning messages to improve the input Java code. For advanced queries, the code excerpts found (labeled with their locations) are returned. Software data and reports are the output for metrics requests, and datasets are returned for queries requesting syntactic and semantic information.

A. PROGQUERY JAVA COMPILER PLUG-IN

ProgQuery modifies the standard Java compiler. It provides a `ProgQueryPlugin` component that can be used with any Java 8+ SDK compiler (`-Xplugin` option). At compilation time, the Neo4j connection string is passed to the plug-in, so that it can store the syntactic and semantic graph representations depicted in Section IV-B.

The Java compiler plug-in interface allows the user to modify and extend its behavior by registering subscribers to various events. Such events occur before and after every processing stage of the compiler, per source file. These events are *parse* (syntax analysis and AST creation), *enter* (source code imports are resolved), *analyze* (semantic analysis) and *generate* (code generation). Our plug-in subscribes to the event that takes place when semantic analysis is finished (*analyze*). Therefore, we intercept the compilation process when the AST has been annotated with the type information inferred by the semantic analyzer [41].

The annotated AST is traversed with the *Visitor* design pattern [31]. We first create the seven overlaid program representations in memory, and later store them into a Neo4j database. We implement four different visitors. `ASTTypesVisitor` translates the Java compiler AST into our AST representation, and creates the Program Dependency Graph (PDG), Class Dependency Graph (CDG) and Call Graph representations; `CFGVisitor` creates the Control Flow Graph (CFG); and `TypeVisitor` and `KeyTypeVisitor` create the Type Graph. The Package Graph is created with a simple traversal of the CDG.

B. OVERLAID PROGRAM REPRESENTATIONS

One of the key features of ProgQuery is the syntactic and semantic information provided as different graph representations. The AST is the core structure, where nodes represent

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

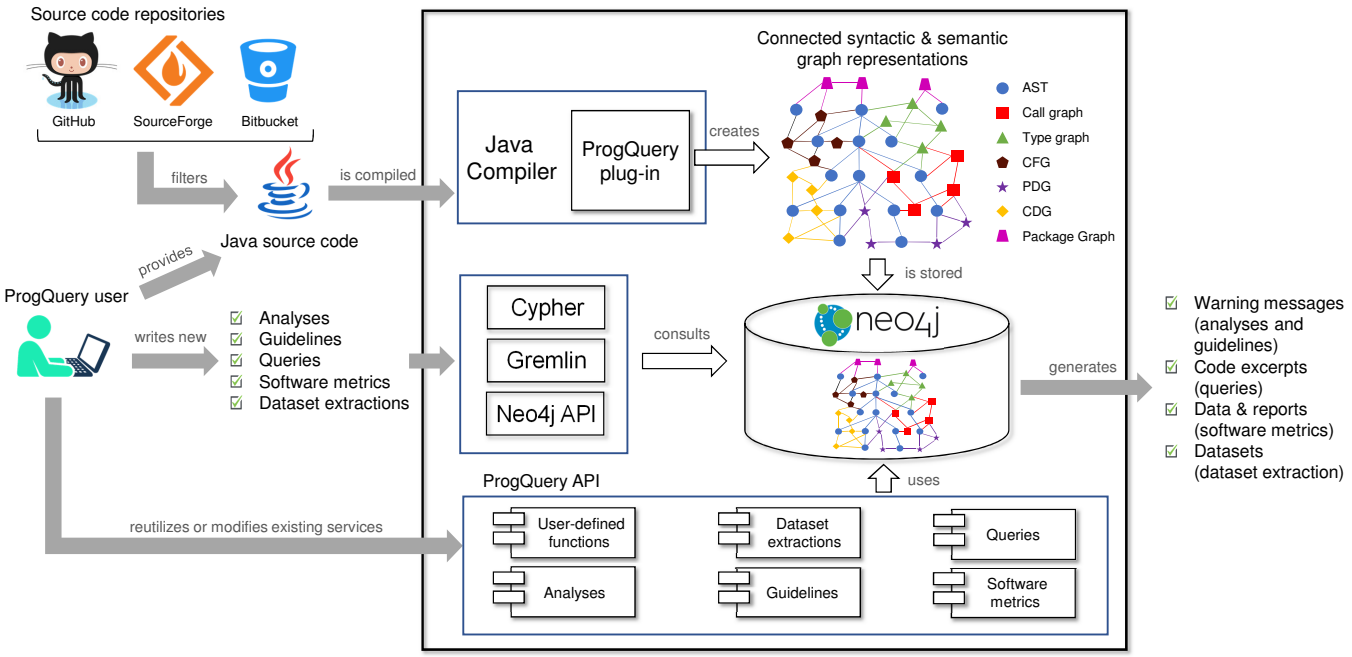


FIGURE 3. Architecture of ProgQuery.

syntactic constructs, hierarchically linked through syntactic relationships. AST nodes are also linked to other (semantic or syntactic) nodes through semantic relationships. Therefore, the syntactic and semantic graph representations are overlaid. This makes it easier to express queries that combine syntactic and semantic information, such as the one in Figure 2.

Figure 4 shows a small excerpt of the seven graph representations created by ProgQuery for the source code in Figure 1. What follows is a brief description of such representations (the whole ontology is detailed in Appendix A).

1) Abstract Syntax Tree (AST)

An AST represents the syntactic information of the input program [1]. For a given program, we create different ASTs where the root nodes are the different compilation units, i.e. Java files. In Figure 4, the nodes n_1 and n_2 represent, respectively, `Polygon.java` and `Figure2D.java`. These two nodes are syntactically connected to the types implemented in each file (`Polygon` and `Figure2D`).

Each AST node representing a type collects, as child nodes, the members defined for that type. For example, the class `Polygon` (n_3) collects its constructor (n_5), the `points` field (n_6), and the `clonePoints` (n_7) and `getPerimeter` (n_8) methods. Child nodes of methods and constructors include their bodies (e.g., the `BLOCK` n_{10} node), parameters (n_{11}), and `throws` clauses. Method bodies hold collections of statements (`if` statement of n_{12}, and `clonePoints` method invocation of n_{14}) that, in turn, may contain other statements (n_{13}) or expressions (n_{15} and n_{16}).

We use Neo4j labels to classify the different AST node types. `COMPILATION_UNIT` and `BLOCK` are two example labels depicted in Figure 4[2]. We also use the multiple label capability provided by Neo4j to allow common generalizations of AST nodes [41]. For example, the assignment expression in n_{16} has the `ASSIGNMENT` label, but also the generalization labels `EXPRESSION`, `AST_NODE` and `PQ_NODE`. This polymorphic generalization design supported by the multiple label feature is valuable to improve the expressiveness of ProgQuery (Section V-C).

Syntactic relationships between nodes are also labeled in Neo4j. For example, node n_1 in Figure 4 is connected to node n_3 through a `HAS_TYPE_DEF` labeled relationship[2].

2) Control Flow Graph (CFG)

CFGs represent the execution paths that *may* or *must* be traversed when the program is run. First, it connects method or constructor definitions to their first statement (e.g., `CFG_ENTRIES` connects the n_5 constructor to n_{12}, its initial statement). When there is no jump, a statement is connected to the following one with a `NEXT_STMT` relationship. For example, the `clonePoints` method invocation in line 15 of Figure 1 (n_{14} in Figure 4) is connected to a CFG semantic node n_{18} through `NEXT_STMT`. The semantic node n_{18} represents the end of the non-exceptional execution of a method (`CFG_NORMAL_END`).

Different control flow statements (e.g., `if`, `while` and `for`) involve a jump in the execution flow. For such cases, the `NEXT_STMT_IF_TRUE` and `NEXT_STMT_IF_FALSE` relationships represent the conditional changes in the execution flow (an example is the connections between the `if` statement in n_{12} and the n_{13} and n_{14} AST nodes).

[2]We do not show all the labels for the sake of readability; see Appendix A for detailed information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

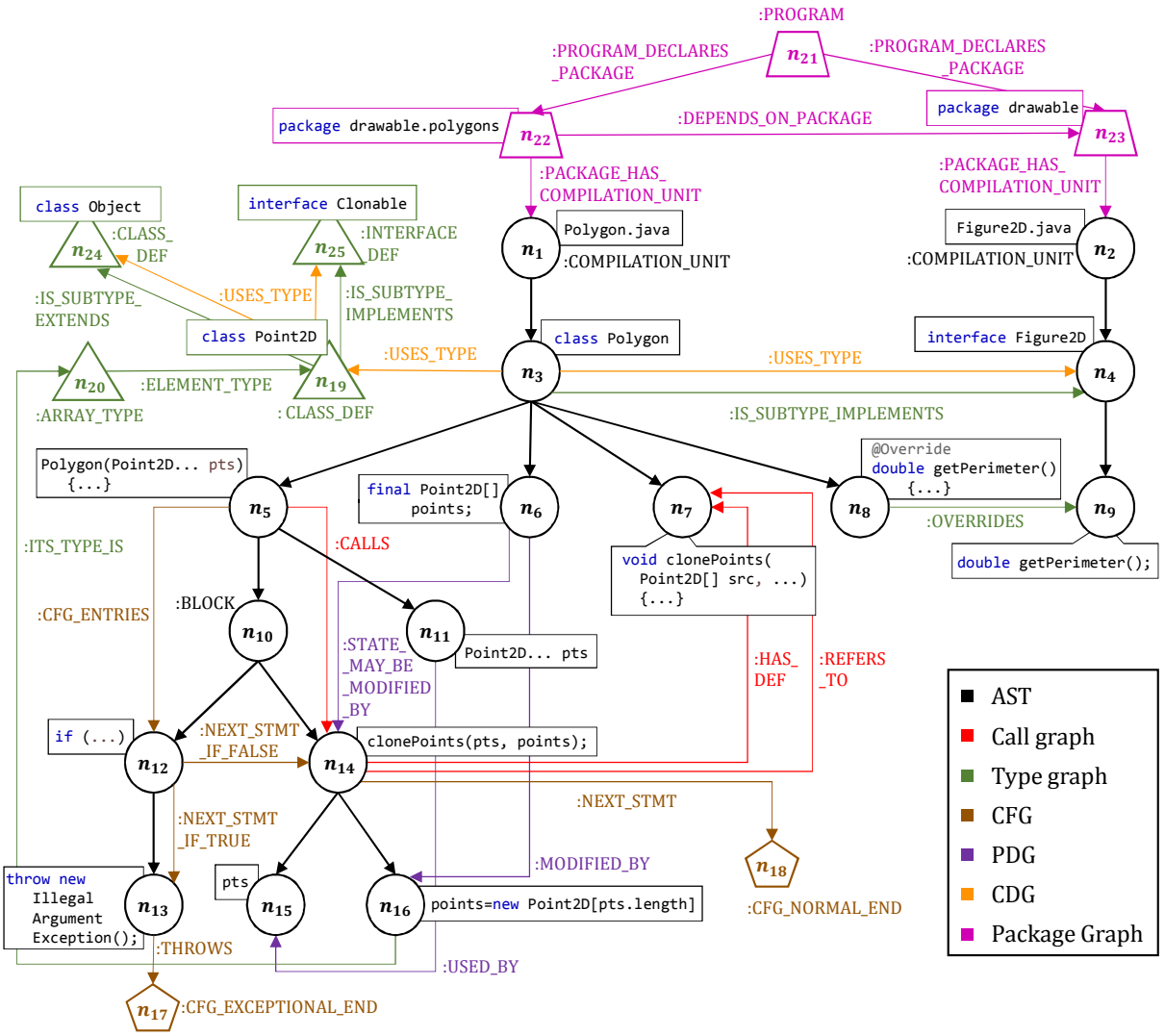


FIGURE 4. Seven different graph representations for the Java program in Figure 1.

CFG also models the exceptional jumps performed by Java checked exceptions[3], and `assert` and `throw` instructions. For that purpose, ProgQuery defines CFG nodes representing exceptional method termination, and exceptions handled in a `catch` or `finally` block. For exception handling, the static 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 n_{13} is connected to the `CFG_EXCEPTIONAL_END` node n_{17} through a *must* `THROWS` relationship, because no `catch` or `finally` blocks are used to handle the exception.

3) Program Dependency Graph (PDG)

PDGs in ProgQuery provide information about when variables (fields, parameters and local variables) and the state of the object they point to (in case variables are references) are read or modified. In this graph, most relationships represent *must* information, meaning that, if a statement is executed, it will certainly read/modify a variable[4]. In Figure 4, the `pts` parameter in n_{11} is read by (`USED_BY`) the `pts` expression in n_{15}, passed as the first argument to the `clonePoints` invocation. Likewise, the `points` field of the example program (n_6) is `MODIFIED_BY` the assignment modeled by the n_{16} AST node.

As mentioned, our PDG representation also provides information about when the state of the object pointed by a reference *may* or *must* be changed. This is a valuable information for many analyses that consider object mutability [36]. For

[3]In Java, unchecked exceptions are `RuntimeException`, `Error` and their subclasses.

[4]This information could be combined with the CFG to know if the statements *must* or *may* be executed.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

that purpose, ProgQuery provides the `STATE_MODIFIED_BY` and `STATE_MAY_BE_MODIFIED_BY` relationships. The former connects a variable to a statement or expression that certainly modifies the state of the object. The latter represents that object modification *may* occur, depending on the execution flow. Moreover, this information is inter-procedural, meaning that method bodies are analyzed to see if they could modify the state of their parameters. For instance, the object pointed by `points` (n_6) *may* be modified in the `clonePoints` invocation (n_{14}), because `points` is passed as the second argument and it could be modified in the `for` loop inside `clonePoints` (line 21, Figure 1).

4) Call Graph

This representation provides information about method and constructor invocations in the source code. Method/constructor definition nodes are connected through `CALLS` relationships to all the invocations in their bodies. In Figure 4, the constructor represented by n_5 is linked to the `clonePoints` invocation in n_{14}. Method/constructor invocations are also linked with their static definition through a `HAS_DEF` relationship (e.g., one connection between n_{14} and n_7). In this way, Call Graphs allow the user to easily navigate through the different method/constructor definitions and invocations in the code.

Call Graphs in ProgQuery also consider polymorphic invocations. When the invoked method has been overridden, all the method implementations that may be called are obtained with the `MAY_REFER_TO` relationship. On the contrary, the `REFERS_TO` arc connects a method invocation to the only possible method definition that could be invoked (e.g., the second link between n_{14} and n_7).

5) Type Graph

Type graphs represent all the type information in the source program. All the expression nodes are linked with their static type. For example, the assignment expression represented by n_{16} is connected to its type through the `ITS_TYPE_IS` relationship. The type is represented with a new n_{20} node of the type graph. n_{20} represents an array type, linked to its `ELEMENT_TYPE` (`Point2D`). In ProgQuery, all the instances of the same type are represented with the same node in the type graph.

Type graphs also model hierarchical relationships among types, making it easy to traverse classes, interfaces and enumerations in the source program. For that purpose, ProgQuery provides the `IS_SUBTYPE_IMPLEMENTS` and `IS_SUBTYPE_EXTENDS` relationships (the `Polygon` class of our example, n_3, implements the `Figure2D` interface, n_4). Information about method overriding is also offered. The `OVERRIDES` relationship links method implementations with the overridden method definitions (if any)—e.g., connection between n_8 and n_9.

`Point2D` is not a class defined by the programmer. It belongs to the standard `java.awt.geom` Java package, so its source code is not included in the program representation. This is the reason why ProgQuery creates the new n_{19} node as part of the Type Graph, not included in the AST. Therefore, ProgQuery provides information not only for the source code, but also for the standard types used in the program. For example, n_{19}, n_{24} and n_{25} represent the Java `Point2D`, `Object` and `Cloneable` types, not defined in the source program. For these three types, ProgQuery sets to false their `isDeclared` property.

6) Class Dependency Graph (CDG)

The CDG representation is aimed at defining the usage relationships among types. The only relationship provided is `USES_TYPE`, which connects two type (class, interface or enumeration) definitions. T_1 is connected to T_2 when T_1 somehow depends on T_2. This dependence means that T_1 defines a local variable, field or parameter of type T_2, extends or implements T_2, defines a method returning T_2, etc. In our example, `Polygon` uses `Figure2D` and `Point2D`. Likewise, `Point2D` uses `Object` and `Cloneable`.

7) Package Graph

The Package Graph gives information about package dependency, and joins up all the ASTs in a program. This representation creates a new package node for each package defined in the source code (n_{22} and n_{23} in Figure 4), which is in turn linked to their compilation units (n_1 and n_2). The `DEPENDS_ON_PACKAGE` relationship links two packages when one depends on the other. This dependency relationship is derived from the `USES_TYPE` class dependency defined in the CDG.

ProgQuery also creates a root node for each program inserted in the system. The n_{21} `PROGRAM` node in Figure 4 represents the root node for the source code in Figure 1. This node is connected through `PROGRAM_DECLARES_PACKAGE` to all the package nodes defined in the program (n_{22} and n_{23}). With this design, each program is modeled with a graph built with seven different overlaid representations, where `PROGRAM` can be thought as the root node of the combined AST.

V. EVALUATION

In this section, we evaluate our platform and compare it with related approaches. We conduct various experiments to address the following research questions about our system:

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?

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

7) Are there any drawbacks of our system, compared to related approaches?

A. METHODOLOGY

We present the selected programs, analyses, and systems to be measured in our experiments. Afterwards, we describe how execution time and runtime memory consumption is measured. We then depict how we configured each system to run the experiments.

1) Programs used

We took the programs to be analyzed from the GitHub Java corpus collected by the CUP research group of the University of Edinburgh [42]. This corpus provides 14,735 projects with different sizes, taking all the GitHub Java projects with at least one fork. This code has already been used in other research works, such as learning coding conventions [43], API mining [44], and database framework usage analysis [45].

We want to use programs of different sizes to study how the evaluated systems scale to increasing program sizes. For that purpose, we sorted the programs by their number of non-empty lines of code and divided them into nine parts, taking one program from each group. In this way, we have nine programs of different sizes. Table 1 shows the selected programs, a brief description, their percentile in the CUP corpus, and the number of non-empty lines of code. Considering all the Java programs in GitHub, the biggest program selected (NFEGuardianShared) is at the 92^{th} percentile.

2) Analyses implemented as queries

We took different analyses from the Java coding guidelines collected by the CERT division of the Software Engineering Institute at Carnegie Mellon University [18]. The CERT division is aimed at improving security and resilience of computer systems [46]. Among other tasks, CERT identifies coding practices that can be used to improve the quality of software systems, classifying them as rules or recommendations [47]. These coding practices are taken from different programming language experts and other sources such as [36], [48] and [49]. Those recommendations are later discussed and revised by the programming community [18].

The CERT Java coding catalog contains more than 83 recommendations, divided into five different categories: programming misconceptions, reliability, security, defensive programming and program understandability. We selected 13 recommendations to code them as static analyses, choosing at least two recommendations from each category. The following enumeration describes each recommendation, its category and identifier, and its original source.

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 of any Java program, implying potential vulnerabilities. We detect types implementing `Serializable` with non-transient fields derived from system resources such as `File`, `NamingContext` and `DomainManager`.
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.
6) DCL60-J (defensive programming) [49], [51]. *Avoid cyclic dependencies between packages*. Cyclic dependencies cause issues related with testing, maintainability, reusability and deployment.
7) OBJ54-J (programming misconceptions) [36]. *Do not attempt to help the garbage collector by setting local reference variables to* `null`. This analysis checks the assignment of `null` to local variables that are no longer needed.
8) OBJ50-J (programming misconceptions) [36], [48]. *Never confuse the immutability of a reference with that of the referenced object*. It is checked that the states of objects pointed by `final` references are not modified (example in Section II).
9) ERR54-J (reliability) [48]. *Use a try-with-resources statement to safely handle closeable resources*. We detect when a variable that implements `AutoCloseable` is not initialized in a try-with-resources statement, and the code may throw an exception before calling `close`. In that case, a try-with-resources statement is advised to the programmer.
10) MET52-J (security) [52]. *Do not use the* `clone` *method to copy untrusted method parameters*. Inappropriate implementations of the `clone` method return objects that bypass validation and security checks. That vulnerable implementation of `clone` is commonly hidden by the attacker in derived classes of the cloned parameter. Thus, the analysis checks when `clone` is invoked against a parameter in a public method of a public class, and the type of the parameter is not `final` (overridable).
11) DCL53-J (defensive programming) [36]. *Minimize the scope of variables*. We search for fields that are unconditionally assigned before their usage, for all the methods in their classes. The analysis encourages the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

TABLE 1. Programs selected from the CUP GitHub Java corpus (percentile and position refer to the non-empty lines of code).

Percentile	Program	Non-empty LoC	Program position in corpus
8	Accelerometer-for-Android: Android app to show real-time accelerometer data.	185	1211
17	Yschool-mini-jeyakaran: Java web application to teach Java programming.	369	2410
25	Clustersoft: Android app to manage remainders.	618	3597
42	Arithmetic-expression-evaluator: scanner, parser and interpreter of arithmetic expressions.	1,404	5962
50	Thumbslug: proxy application to manage certificates of Candlepin software subscriptions.	2,086	7168
58	Comm: socket component to handle communications of the S4 streaming computing platform.	3,119	8358
75	OpenNLP-Maxent-Joliciel: supervised machine-learning application for natural language processing.	7,936	10747
83	Frankenstein: testing framework for Swing GUI Java applications.	15,308	11951
93	NFEGuardianShared: Web repository for XML electronic invoices.	55,789	13366

programmer to use local variables instead.

12) OBJ56-J (security) [53]. *Provide sensitive mutable classes with unmodifiable wrappers*. When a given class is mutable because of m modifier methods, it is checked that one derived class provides an immutable wrapper. In such wrapper, those m methods must be overridden with implementations where the state of the object is not modified.
13) NUM50-J (program understandability) [48]. *Convert integers to floating point for floating-point operations*. The analysis checks division expressions where the two operands are/promote to integers, and the result is assigned to a `float` or `double`. In that scenario, there might be loss of information about any possible fractional remainder.

We implemented these 13 program analyses in ProgQuery (the Cypher code can be download from [54]). Therefore, the answer to Research Question 1 (Section V) is that the ontology defined in Appendix A can be used to express real program analyses.

3) Systems measured

We included in our evaluation different systems related to our approach. The analyses described in the previous subsection were coded for such systems in order to compare them. These are the selected systems:

- Wiggle 1.0 [9], a source-code querying system based on a graph data model. Wiggle represents programs as graph data models, and stores them in Neo4j. The Cypher graph query language is used to express advanced queries, including syntactic (mainly) and some semantic properties of programs. We use Neo4j Community 3.5.6 server and Neo4j 3.3.4 embedded. The former mode provides direct use from the Java client, loading the database engine in the same JVM process as the client application. The latter mode runs Neo4j as a separate process (DataBase Management System, DBMS) via RESTful services [55].
- Semmle CodeQL 1.20, a code analysis platform to perform detailed analyses of source code [21]. Semmle allows writing queries in QL, an object-oriented variant of the Datalog logical query language [56]. Semmle CodeQL stores programs in a PostgreSQL relational database. It promotes variant analysis, the process of using a known vulnerability as a seed to find similar problems in the code [57]. Semmle provides a set of existing analyses to facilitate the variant analysis approach.
- ProgQuery 1.1. We include the latest version of our system in the evaluation. ProgQuery is measured with the same two Neo4j versions we used to measure Wiggle: Neo4j Community 3.5.6 server and Neo4j embedded 3.3.4.

4) Data analysis

The execution time of a Java program is affected by many factors such as just-in-time (JIT) compilation, hotspot dynamic optimizations, thread scheduling and garbage collection. This non-determinism at runtime causes the execution time of Java programs to differ from run to run. For this reason, we use the statistically rigorous methodology proposed by Georges *et al.* [58]. To measure execution time and runtime memory consumption, a two-step methodology is followed:

1) We measure the execution time of running the same program multiple times. This results in p measurements x_i with $1 \leq i \leq p$.
2) The confidence interval for a given confidence level (95%) is computed to eliminate measurement errors that may introduce a bias in the evaluation. The computation of the confidence interval is based on the central limit theorem. That theorem states that, for a sufficiently large number of samples (commonly $p \geq 30$), $\overline{x}$ (the arithmetic mean of the x_i measurements) is approximately Gaussian distributed, provided that the samples x_i are independent and they come from the same population [58]. Therefore, taking $p = 30$, we can compute the confidence interval $[c_1, c_2]$ using the Student's t-distribution as [59]:

$$c_1 = \overline{x} - t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}} \qquad c_2 = \overline{x} + t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

where $\alpha = 0.05$ (95%); s is the standard deviation of the x_i measurements; and $t_{1-\alpha/2;p-1}$ is defined such that a random variable T, which follows the Student's t-distribution with $p-1$ degrees of freedom, obeys $Pr[T \leq t_{1-\alpha/2;p-1}] = 1-\alpha/2$. In the subsequent figures, we show the mean of the confidence interval plus the width of the confidence interval relative to the mean (bar whiskers). If two confidence intervals do not overlap, we can conclude that there is a statistically significant difference with a 95% $(1-\alpha)$ probability [58].

Memory consumption is measured following the same two-step methodology. Instead of measuring execution time, we compute the maximum size of working set memory used by the process since it was started (the `PeakWorkingSet` property) [60]. The working set of a process is the set of memory pages currently visible to the process in physical RAM memory. The `PeakWorkingSet` is measured with explicit calls to the services of the Windows Management Instrumentation infrastructure [61].

All the tests were carried out on a 2.10 GHz Intel(R) Xeon(R) CPU E5-2620 v4 (6 cores) with 32GB of RAM running a 64-bit version of Windows 10.0.18362 Professional. We used Java 8 update 111 for Windows 64 bits. The benchmarks were executed after system reboot, removing the extraneous load, and waiting for the operating system to be loaded (until the CPU usage falls below 2% and remains at this level for 30 seconds). To compute average percentages, factors and orders of magnitude, we use the geometric mean.

5) Experimental environment

To measure the systems described in Section V-A3, various configuration variables need to be provided (e.g., heap memory size of Neo4j, number of insertions per transaction, and heap memory used by the Java virtual machine). Runtime performance of those systems depend on the values of such variables. Therefore, we need to find the values for which the selected systems perform optimally, in the above mentioned computer.

We followed the following algorithm to find the optimal values for the configuration variables. First, we fix all the variables to their default values. Then, for each variable, we analyze the influence of that variable on the system performance. We select the value when the system converges to its best performance. This process is repeated for all the variables, until the system performance cannot be further optimized.

Figure 5 shows an example of how two variables influence on runtime performance of the system. In the left-hand side, it is showed how the heap size of our modification of the Java compiler influences on insertion time (Neo4j embedded). The right-hand side illustrates how insertion times in Neo4j server depend on the number of insertions per transaction done by ProgQuery and Wiggle. Orange lines in Figure 5 indicate the execution time with the default values; red dots specify the value chosen by our algorithm. We can see how runtime performance is improved in both scenarios.

The following enumeration lists the variables that influence on the performance of the selected systems, and the values we get by applying our algorithm:

- Neo4j page cache size (`dbms.memory.pagecache.size`): 16.06 MB for insertion and analysis.
- Number of insertions per transaction (`operationsPerTransaction`): 500 insertions per transaction.
- For insert operations, the startup memory used by our modified Java compiler (`-J-Xmx` and `-J-Xms`): 2 GB.
- For executing queries, maximum (`-Xmx`) and initial (`-Xms`) heap memory used by the Java Virtual Machine: we set both variables to 1 GB.
- Memory for running queries of Semmle CodeQL: 1 GB.
- Initial (`dbms.memory.heap.initial_size`) and maximum (`dbms.memory.heap.max_size`) heap memory size used by Neo4j. For both variables, we choose 500 MB for insertion operations and 1 GB for analyses (queries).

B. ANALYSIS TIME

We measure and compare runtime performance of the systems described in Section V-A3. Their scalability related to program size and analysis complexity is also discussed.

1) Increasing program sizes

The first comparison, illustrated by Figure 6, presents execution time of analyses for increasing program sizes. We take the size of a program to be the number of its nodes plus the number of its arcs, using *Wiggle*'s AST representation. The values shown are the average times for all the analyses in Section V-A2, relative to *ProgQuery embedded*.

For all the programs, *ProgQuery server* outperforms the rest of systems. It is 9.2 times faster than its embedded version. On average, *ProgQuery server* performs, respectively, 48, 53 and 245 times better than *Semmle*, *Wiggle server* and *Wiggle embedded*.

To analyze the scalability of the systems, Figure 7 displays the absolute execution time trends when program sizes increase. All the execution times grow as program sizes increase, but *ProgQuery* is the system with the lowest growth. *ProgQuery server*, *ProgQuery embedded*, and *Semmle* show linear scalability (p-values of linear regression models are lower than 0.01), and their slopes are, respectively, 16, 29 and 929.

Figure 7 shows how *Wiggle* performs better than *Semmle* for shorter programs, but its execution time grows significantly higher than *Semmle*, as program size increases. It seems that the database start-up cost causes an initial performance penalty, more noticeable for small programs. Moreover, the additional semantic information stored by *Semmle* may cause a decrease in the number of accesses to the database, providing better results when analyzing bigger programs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

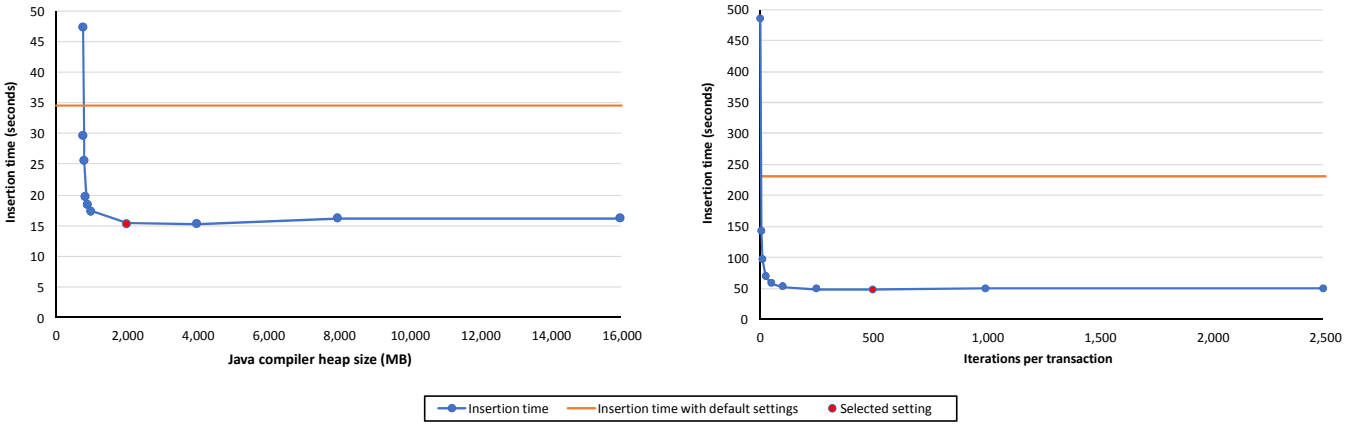


FIGURE 5. Dependency of Java compiler memory (left) and iterations per transaction (right) on insertion time for Neo4j embedded (left) and server (right), when inserting the Frankenstein program in Table 1.

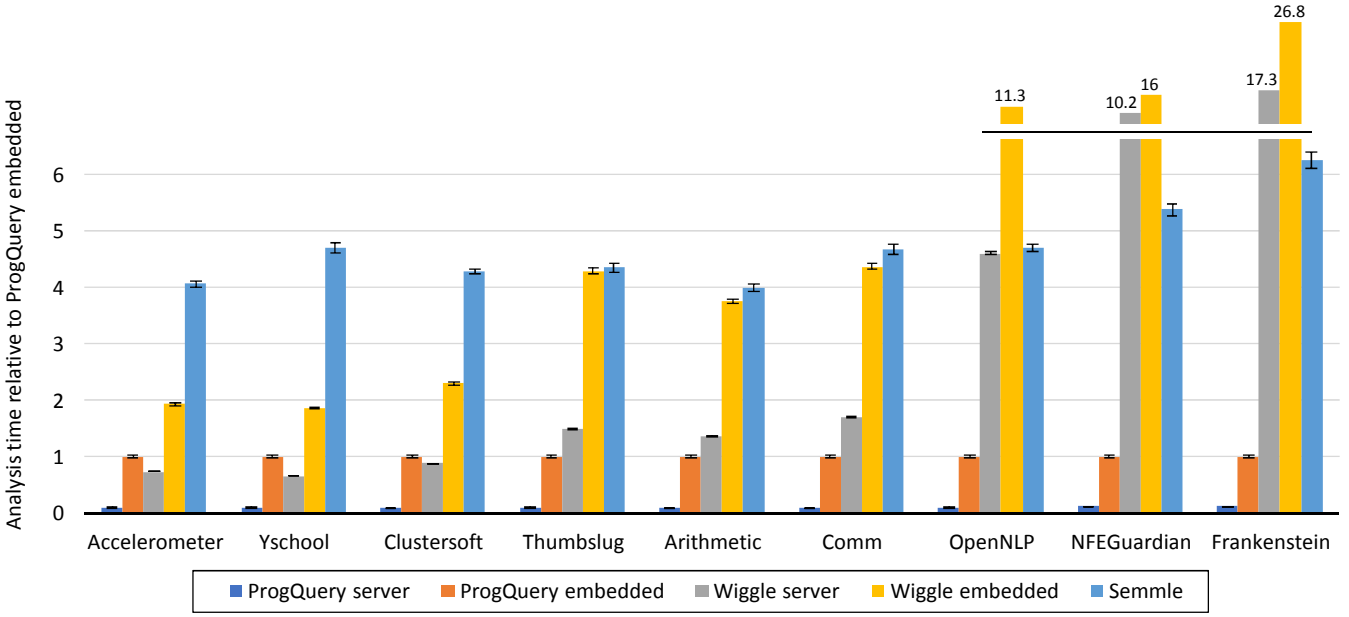


FIGURE 6. Average analysis execution time for increasing program sizes (execution times are relative to *ProgQuery embedded*).

Even though *Wiggle* and *ProgQuery* use the same persistence system (Neo4j), they show significantly different scalabilities. This is caused by the different information stored by the two systems. Since *ProgQuery* stores more semantic information for each program, it reduces the cost of computing that information at runtime. This cost becomes more important as program size grows.

These data provide a response to Research Question 3 (Section V): *ProgQuery* scales significantly better than the other systems to increasing program sizes (we answer Research Question 2 in the following subsection).

2) Increasing complexity of analyses

For each system, we study how the analysis complexity influences on its execution time. We estimate the complexity of each analysis by counting the number of different program representations (Section IV-B) consulted in the analysis.

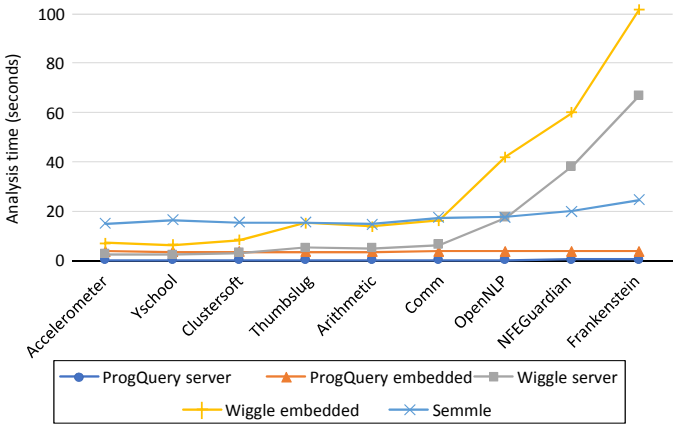


FIGURE 7. Execution time (seconds) trend for increasing program sizes (values shown are the geometric mean of execution times for all the analyses executed against the given program).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

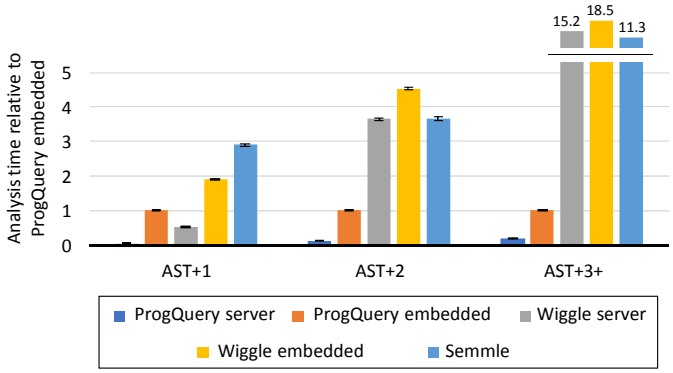


FIGURE 8. Average execution time for increasing analysis complexity (execution times are relative to *ProgQuery embedded*).

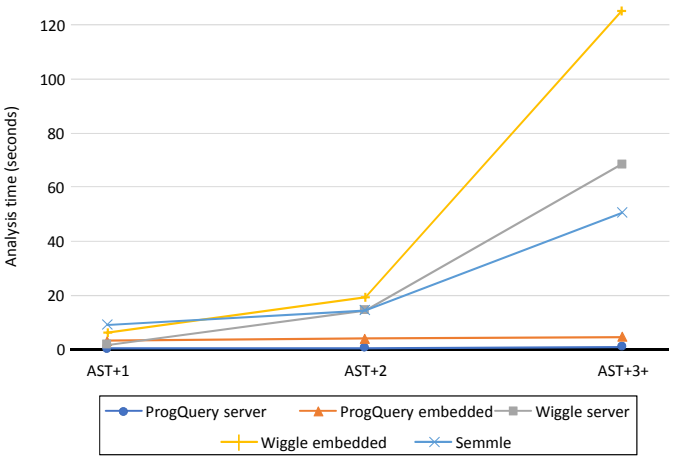


FIGURE 9. Execution time (seconds) trend for increasing analysis complexity (values shown are the geometric mean of execution times of all the programs of the given complexity).

Therefore, we identify three levels of complexity:

- AST+1 (easiest analyses). These analyses use the syntactic representation (AST) and at most another semantic representation.
- AST+2 (medium level of complexity). The AST plus two other semantic representations are queried in these analyses.
- AST+3+ (most complex analyses). These analyses use the AST and three or more semantic representations.

Table 2 shows the program representations used by each analysis and the level of complexity assigned. All the analyses use the AST, since they all start by consulting syntactic information. The analysis that uses most representations (DCL60-J) consults five out of seven.

Figure 8 shows the execution times for the three levels of complexity identified. Values shown are the average analysis times for all the programs in Section V-A1. *ProgQuery server* is the system with the best performance, for all the levels of complexity—in fact, *ProgQuery server* shows the lowest execution times for each single analysis executed. Moreover, *Wiggle server* is the only system that performs better (54%) than *ProgQuery embedded*, only for the easiest (AST+1) analyses. For the remaining cases, *ProgQuery embedded* outperforms the other systems.

Therefore, the answer to Research Question 2 (Section V) is that *ProgQuery server* analysis times are significantly lower than the existing systems, for all the analyses measured. This response also holds for the embedded version, if we compare it with systems that use an embedded database instead of a DBMS (i.e., *Wiggle embedded* and *Semmle*).

Figure 9 shows how execution time depends on the complexity level of analyses for all the systems. Analysis time grows as complexity increases. The slope of analysis time growth (linear regression) for *Semmle* and *Wiggle* is, respectively, 1.76 and 1.97 orders of magnitude higher than *ProgQuery*. Therefore, the answer to Research Question 4 (Section V) is that *ProgQuery* scales significantly better than *Semmle* and *Wiggle* for analysis complexity.

For the easiest analyses, *Wiggle* performs better than *Semmle*. For complex queries, however, it is the other way around. Since most of the information in *Wiggle* is syntactic, easy analyses just consult information in the database (most of the information searched belongs to the AST). As more semantic information is required, *Wiggle* analyses need to compute that information from the existing one, producing a runtime performance cost. In *ProgQuery*, that computation is not necessary, because it provides more semantic representations (Section IV-B). Finally, it seems that *Semmle*'s logical query language QL manages to reduce the number of accesses to the relational database in complex analyses, performing better than *Wiggle*.

3) Limit values

We also measure the systems' capability to perform analyses against huge programs. By using the Google's BigQuery project [10], we identified the biggest Java project in GitHub (Medicayundicom), which has 18M lines of code. Then, we measure analysis time for all the analyses in Section V-A2 against such a huge project. Since Medicayundicom does not provide maven `pom` files to compile the project (a requirement of all the systems evaluated), we combine different existing Java projects into a single one, until the new project reaches 18M lines of code.

ProgQuery is the only system that runs all the analyses for that program. Both *Semmle* and *Wiggle* prompted memory errors at insert or analysis time, and hence analyses could not be run. Table 3 shows *ProgQuery* analysis times for all the queries, under the configuration settings described in Section V-A5. That table gives us the response to Research Question 5 (Section V): *ProgQuery* is able to run all the analyses measured against a huge program with an average execution time of 53 seconds, while most of the analyses are executed in tens of seconds.

We can see in Table 3 that there is not a strong correlation between *ProgQuery* analysis complexity and execution time. This is due to two factors. First, *ProgQuery* provides much semantic information that does not need to be computed.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

TABLE 2. Program representations used by the different analyses and their level of complexity.

Analysis	AST	Call Graph	Type Graph	CFG	PDG	CDG	Package Graph	Complexity
DCL56-J	×							AST+1
MET50-J	×							AST+1
MET52-J	×				×			AST+1
MET55-J	×		×					AST+1
NUM50-J	×				×			AST+1
SEC56-J	×		×					AST+1
MET53-J	×	×	×					AST+2
OBJ54-J	×			×	×			AST+2
OBJ56-J	×		×		×			AST+2
DCL53-J	×		×	×	×			AST+3+
DCL60-J	×		×		×	×	×	AST+3+
ERR54-J	×		×	×	×			AST+3+
OBJ50-J	×	×	×		×			AST+3+

TABLE 3. *ProgQuery* execution time (seconds) for analyses in Section V-A2, run against a Java program of 18M lines of code.

Analysis Name	Complexity	Execution time (seconds)
DCL56-J	AST+1	35.6
MET50-J	AST+1	82.4
MET52-J	AST+1	7.9
MET55-J	AST+1	130.8
NUM50-J	AST+1	38.3
SEC56-J	AST+1	7.9
MET53-J	AST+2	39.0
OBJ54-J	AST+2	39.1
OBJ56-J	AST+2	365.4
DCL53-J	AST+3+	172.8
DCL60-J	AST+3+	36.6
ERR54-J	AST+3+	211.6
OBJ50-J	AST+3+	38.1

Second, the overlaid representations offer interconnected nodes of different kinds, so that queries can combine different information with no additional performance cost. Therefore, execution time depends on the number of nodes consulted (of any representation), rather than on the kind of information consulted.

C. PROGRAM ANALYSIS EXPRESSIVENESS

Both *Wiggle* and *ProgQuery* use Cypher, a declarative graph query language, originally intended to be used with the Neo4j database [19]. Its design is focused on providing the power of SQL, but applied to databases built upon the concepts of graph theory. Cypher, together with PGQL and G-Core, represent the baseline for GQL, the upcoming ISO standard graph query language [62].

Semmle provides the QL object-oriented declarative query language [56]. Its syntax is similar to SQL, but its semantics is based on Datalog, a declarative logic programming language [22]. QL is the way *Semmle* provides graph-based abstractions, since graphs are translated into a relational database. On the contrary, *Wiggle* and *ProgQuery* store graph representations directly in a Neo4j database, avoiding that impedance mismatch [24]. Since graph abstractions are maintained in the persistent storage, the user may utilize different mechanisms to access such graph program representations. For example, Neo4j can be accessed with mechanisms other than Cypher, such as the Gremlin programming language and the Neo4j traversal framework Java API.

Table 4 presents the number of tokens (lexical elements), AST nodes and lines of code of the queries used to write all the analyses for the three different systems. These measures are an estimate of how much code is needed to write the different analyses in each system. We can see how *ProgQuery* is the system that needs less code to express the analyses. Even though *ProgQuery* and *Wiggle* use the same language (Cypher), on average *ProgQuery* requires 18% the code used by *Wiggle*. Moreover, the code in *Semmle* is almost 2 times the code in *ProgQuery*.

These differences among analysis code are mainly caused by the information stored by the systems. Since *ProgQuery* stores seven different program representations (Section IV-B), analyses do not need to use additional code to compute such information, reducing the length the code. This fact also causes that the source code of *ProgQuery* analyses is not increased from medium (AST+2) to complex (AST+3+) complexities, unlike the other systems (Table 4).

After this study about expressiveness, we can answer Research Question 6 (Section V). *ProgQuery* provides an expressive and declarative mechanism to express program analysis. Although no standard graph query language exists yet, the language used by *ProgQuery* represents a strong influence on the upcoming standard.

D. MEMORY CONSUMPTION

Figure 10 shows the average RAM memory consumed by the five systems evaluated, under the configuration settings described in Section V-A5. *ProgQuery embedded* is the system that consumes fewer resources (*Wiggle embedded* and *Semmle* consume 22% and 42% more memory). The extra semantic information stored by *ProgQuery* in the database causes the lower consumption of RAM memory resources, which the other systems use to compute such information. The same difference appears when we compare the two Neo4j server systems: *Wiggle* consumes 5% more memory

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

TABLE 4. Number of tokens (lexical elements), AST nodes, and lines of code of the queries used to write all the analyses in the different systems.

		Tokens			AST nodes			Lines of Code		
	Analysis	**ProgQuery**	**Semmle**	**Wiggle**	**ProgQuery**	**Semmle**	**Wiggle**	**ProgQuery**	**Semmle**	**Wiggle**
AST+1	DCL56-J	70	93	359	30	49	123	3	4	13
	MET50-J	190	202	453	95	115	181	6	8	17
	MET52-J	173	166	351	74	81	180	5	15	11
	MET55-J	159	154	590	67	77	248	7	11	21
	NUM50-J	221	292	551	104	160	270	5	34	14
	SEC56-J	112	157	589	49	78	253	5	15	34
	Mean (AST+1)	**144**	**167**	**471**	**65**	**87**	**202**	**5**	**12**	**17**
AST+2	MET53-J	139	192	1,415	70	104	661	6	24	41
	OBJ54-J	127	348	1,316	58	177	645	5	31	42
	OBJ56-J	772	783	2,439	395	403	1,265	25	48	54
	Mean (AST+2)	**239**	**374**	**1,656**	**117**	**195**	**814**	**9**	**33**	**45**
AST+3+	DCL53-J	507	878	1,195	253	457	638	15	74	35
	DCL60-J	77	380	1,467	35	201	644	3	43	44
	ERR54-J	691	990	5,528	335	538	2,765	18	110	275
	OBJ50-J	126	1,392	3,061	59	757	1,533	5	142	75
	Mean (AST+3+)	**241**	**823**	**2,334**	**115**	**440**	**1,149**	**8**	**84**	**75**
	Mean (total)	**190**	**329**	**1,030**	**88**	**173**	**476**	**7**	**27**	**34**

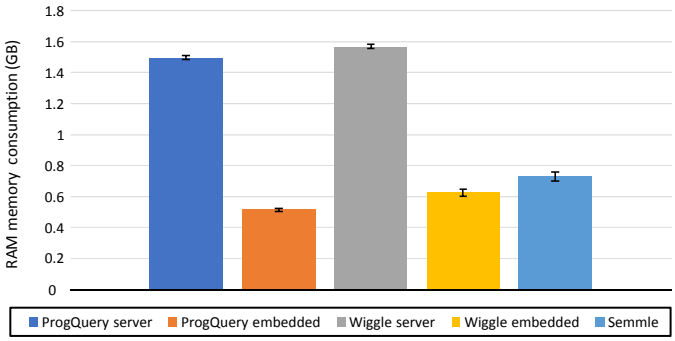


FIGURE 10. RAM memory consumed at analysis execution.

than *ProgQuery*. The difference between *ProgQuery server* and *Semmle* (almost one factor) is caused by the memory consumption of Neo4j server, when compared to its embedded version: 1 GB.

Since *ProgQuery* stores more information in the database that the other systems, we also compare the sizes of each database. The average database sizes per program for *ProgQuery* are, respectively, 25% and 97% greater than *Semmle* and *Wiggle*.

E. INSERTION TIME

ProgQuery provides runtime performance and expressiveness benefits mainly because of the additional program representations stored for each program. However, the process of computing that supplementary information at compile time plus its storage in the database involve extra insertion time. For this reason, we measure insertion time for all the systems but *Semmle*. We could not measure *Semmle* because it only provides insertion through its LGTM web application [63]. The user provides a code repository identifier, and LGTM compiles the program, returning the relational database file containing the program representation.

In Figure 11, we can see how *ProgQuery server* is the system with the highest insertion time: on average, 60% more time than the same version for *Wiggle*. This difference is caused by the additional program representations stored in our system. However, as program size grows, the difference between *ProgQuery server* and *Wiggle server* decreases. This is because of a separate optimization we employ at compile time, based on avoiding the use of reflection [64] performed by *Wiggle* (Section III).

For the embedded versions, *ProgQuery* performs slightly better than *Wiggle*. Average insertion times for *Wiggle* are 6% higher. Our optimization makes *ProgQuery* perform better, even though it inserts more nodes and arcs than *Wiggle*.

Figure 11 also displays compilation time used by the original Java compiler (without our plug-in). By comparing that value with the rest of measures, we can estimate the time each system needs to create the representations and store them in the database. Compilation time represents 17.8%, 11.8%, 16.8% and 7.4% of the overall insertion time for, respectively, *Wiggle embedded* and *Wiggle server*, and *ProgQuery embedded* and *ProgQuery server*.

Figure 12 shows insertion times per node or arc stored. This figure enables us to analyze the performance of each system relative to the information stored. We can see how our optimization makes our server to be more efficient than *Wiggle* (18% faster for server and 101% for embedded), when measuring insertion time per element stored. Neo4j server seems to perform additional operations at insertion to optimize queries, so our optimization is more evident in the embedded version than in the server one.

Research Question 7 (Section V) can be answered after analyzing runtime memory consumption and insertion time. The main drawback of *ProgQuery* is the increase of insertion time, but only for the server version. The average insertion time increase is 60%, but this drops as program size grows. Another minor drawback is database size, which shows in-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

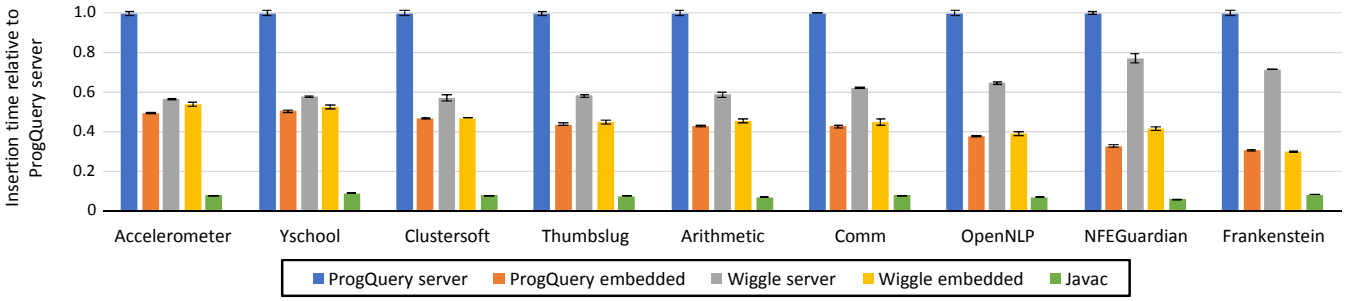


FIGURE 11. Insertion times for increasing program sizes, relative to *ProgQuery server*.

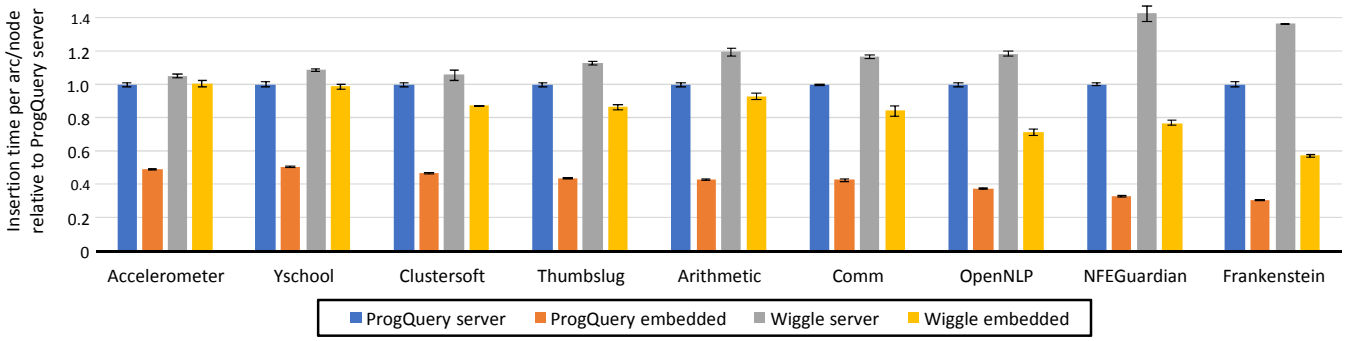


FIGURE 12. Insertion times per node/arc for increasing program sizes, relative to *ProgQuery server*.

creases from 25% up to 97%.

VI. CONCLUSIONS

ProgQuery shows how overlaid graph representations of syntactic and semantic information of Java source code can be used to efficiently perform advanced source code analyses. By storing those graph representations in a Neo4j database, average analysis times range from 48 to 245 times faster than the related systems. Our platform scales better to increasing program sizes and query complexity. Moreover, ProgQuery can analyze huge Java programs in tens of seconds, whereas the other alternatives exhaust memory. For all the analyses coded, ProgQuery seemed to be more expressive than the other systems.

These benefits are partially due to the additional semantic information calculated and stored by our system. The computation and storage of that supplementary information involves 60% insertion time increase, but only when the Neo4j server database is used. Likewise, average database size grows from 25% to 97%. However, these penalties are much lower than the average analysis time benefits obtained (from 5x to 25x).

We are currently developing a web application to offer the services implemented by ProgQuery online. Users may upload their code or take it from an existing repository, and execute their analyses and queries in Cypher. Existing analyses and queries will also be provided. We are also developing various analyses described by Joshua Bloch for effective Java programming [36]. We plan to use ProgQuery for sharing syntactic and semantic program information of existing open-source repositories over the web [65]. ProgQuery will automatically download the source code, compile it, populate a graph database with syntactic and semantic representations, and provide that information online. This can be used for web-based queries, or simply downloaded in XML form. ProgQuery will also be used to extract information of Java programs, and use it to create predictive models for different scenarios [38].

The source code and binaries of our platform, all the analyses and the code corpus used to evaluate the systems, and the assessment data obtained in this work are available for download at http://www.reflection.uniovi.es/bigcode/download/2020/ieee-access

APPENDIX A PROGRAM REPRESENTATIONS

In this appendix, we describe the ontology defined to represent syntactic and semantic information of Java programs.

A. NODES

Figure 13 shows the nodes used for the seven graph representations described in Section IV-B. We use the multi-label capability of Neo4j to assign multiple types (subtyping polymorphism) to a single node. For example, a `METHOD_INVOCATION` node is also classified as `CALL`, `EXPRESSION`, `AST_NODE` and `PQ_NODE`. All the nodes in ProgQuery hold the `PQ_NODE` label. Nodes belonging to AST, Control Flow Graph, Program Dependency Graph, Package Graph and Type Graph are labeled with, respectively, `AST_NODE`, `CFG_NODE`, `PDG_NODE`, `PACKAGE_NODE` and `TYPE_NODE`. The Call Graph and Class Dependency Graph representations define no new nodes (only relationships).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

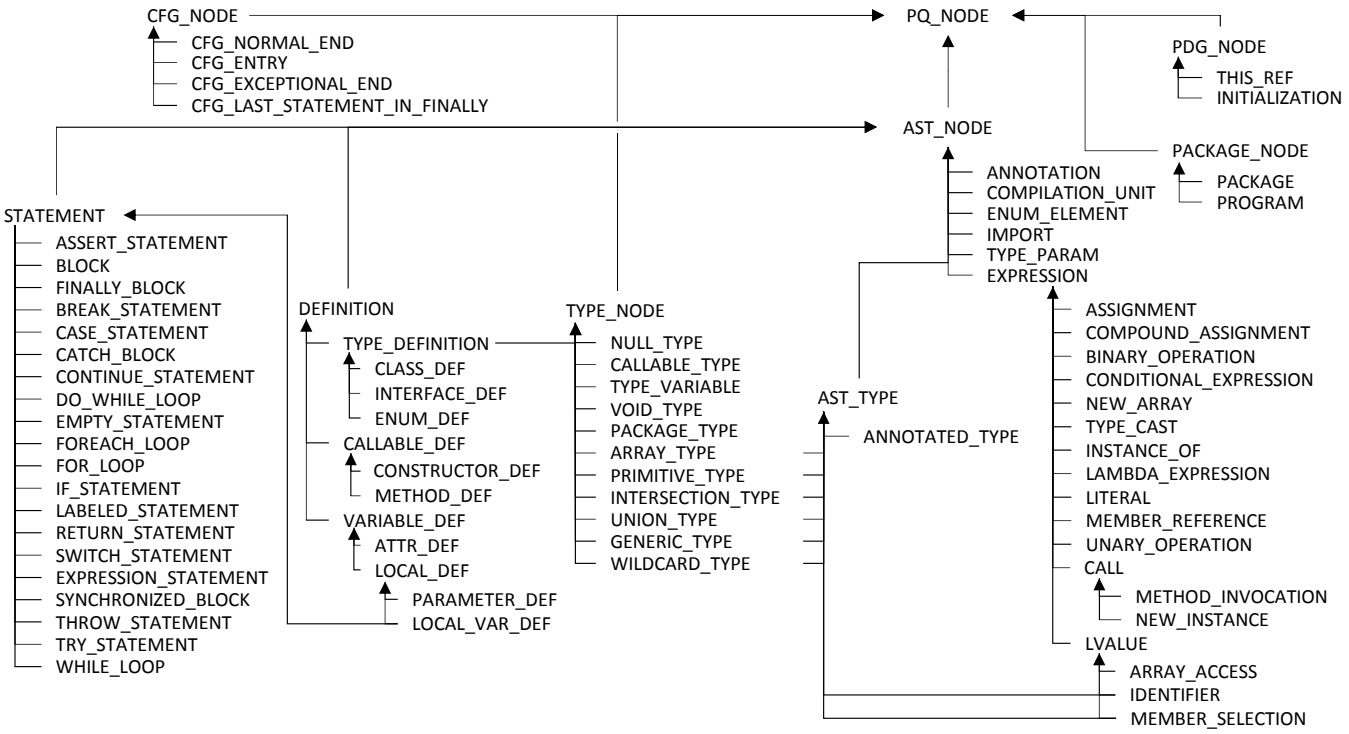


FIGURE 13. Labels defined to categorize the nodes used for the different Java program representations.

B. ABSTRACT SYNTAX TREE

The syntactic information is represented with the AST. This is the main representation in ProgQuery. It provides 67 labels for 56 nodes (Figure 13), 100 relationships and 26 properties. For the sake of readability, we do not include them in this document—they define common syntax elements of an object-oriented language [41]. We detail all the nodes, relationships and properties used to represent the AST in [66].

C. CONTROL FLOW GRAPH

These are the nodes of the CFG:

- `CFG_NORMAL_END`: endpoint of the control flow that represents the normal completion of the method/constructor execution.
- `CFG_ENTRY`: starting point of the control flow connected to the first statement of the method/constructor.
- `CFG_EXCEPTIONAL_END`: endpoint of the control flow; it represents the abrupt completion of the method/constructor execution caused by an exception.
- `CFG_LAST_STATEMENT_IN_FINALLY`: artificial statement created to model the statement just before exiting the `finally` block.

We now describe the relationships of CFG. Table 5 defines their domain (source node), range (target node) and cardinality.

- `CFG_ENTRIES`: relates a callable definition to the entry point of its control flow.
- `CFG_END_OF`: connects the endpoint of the control flow to the method/constructor definition that creates the flow path.
- `CFG_FINALLY_TO_LAST_STMT`: relates a `finally` block to an artificial statement representing the flow just before exiting the `finally` block.
- `CFG_NEXT_STATEMENT`: connects one statement to the following one, when no jump exists.
- `CFG_NEXT_STATEMENT_IF_TRUE`: relates a statement that bifurcates the control flow to the next one, when the condition holds.
- `CFG_NEXT_STATEMENT_IF_FALSE`: relates a statement that bifurcates the control flow to the next one, when the condition does not hold.
- `CFG_FOR_EACH_HAS_NEXT`: relates for-each statements to the first enclosed statement to be executed if there is any element to iterate.
- `CFG_FOR_EACH_NO_MORE_ELEMENTS`: relates for-each statements to the statement outside the loop to be executed if there are no more elements to iterate.
- `CFG_SWITCH_CASE_IS_EQUAL_TO`: relates a `switch` statement to the statement to be executed when a `case` expression is matched.
- `CFG_SWITCH_DEFAULT_CASE`: relates a `switch` statement to the statement to be executed if no `case` expression is matched.
- `CFG_AFTER_FINALLY_PREVIOUS_BREAK`: the last statement in a `finally` block is connected to the statement to be executed in case the `try` block contains a `break` statement.
- `CFG_AFTER_FINALLY_PREVIOUS_CONTINUE`: the last statement in a `finally` block is connected to the statement to be executed in case the `try` block contains a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

`continue` statement.
- `CFG_NO_EXCEPTION`: relates the last statement in a `finally` block to the statement to be executed if no exceptions are thrown.
- `CFG_IF_THERE_IS_UNCAUGHT_EXCEPTION`: relates a `catch` statement or the last statement in a `finally` block to the statement to be executed if a thrown exception is not caught.
- `CFG_CAUGHT_EXCEPTION`: relates a `catch` statement to its enclosed local variable if, considering the hierarchical type information, the exception could be caught.
- `CFG_MAY_THROW`: relates any statement that may throw an exception to statements to be executed if so.
- `CFG_THROWS`: relates a `throw` statement to the statement to be executed after the exception is thrown.

These are the properties of the CFG nodes and relationships (see details in Table 6):

- `mustBeExecuted`: holds whether a statement is unconditionally executed regardless the execution path.
- `exceptionType`: string holding the fully qualified name of the exception type to be thrown.
- `methodName`: string holding the fully qualified name of the method that may raise the checked exception, if any.
- `label`: string holding the label name (if any) of the `break`/`continue` statement that causes the control-flow jump.
- `caseIndex`: integer value representing the index of the `case` (among all the other `case`s contained in the `switch`) to be executed.
- `caseValue`: string representing the expression of the `case` to be executed.

D. CALL GRAPH

No new nodes are defined for the Call Graph. These are the Call Graph relationships (detailed in Table 7):

- `CALLS`: relates a callable definition to the method/constructor invocations in its body.
- `HAS_DEF`: connects invocations to the static definition of the method/constructor invoked.
- `MAY_REFER_TO`: when a method is overridden, this relationship connects the invocation to the method definitions that may be called.
- `REFERS_TO`: when only one method/constructor may be called, `REFERS_TO` connects the call to the definition to be invoked.

The only property defined is `isInitializer` for `CALLABLE_DEF` nodes (one-cardinality and Boolean value-type). It indicates whether a callable definition is an intializer; i.e., it is either a constructor or a (`private` or package) method that is only called from another initializer.

E. TYPE GRAPH

These are the nodes defined for Type Graphs:

- `ARRAY_TYPE`: node representing an array type.
- `TYPE_DEFINITION`: class, enumeration or interface definition.
- `CALLABLE_TYPE`: type of any method or constructor.
- `INTERSECTION_TYPE`: intersection of two or more types (i.e., Java `&` type constructor).
- `VOID_TYPE`: node representing the `void` type.
- `PACKAGE_TYPE`: type attached to an package reference expression (i.e. `java.lang`).
- `NULL_TYPE`: node representing the type of `null`.
- `PRIMITIVE_TYPE`: representation of any Java primitive type.
- `TYPE_VARIABLE`: type variables used with generic types.
- `UNION_TYPE`: union of two or more types, used in `catch` blocks (i.e., Java | type constructor).
- `GENERIC_TYPE`: a generic type that is parameterized with other types.
- `WILDCARD_TYPE`: node representing a Java wildcard type (i.e., `?`).

The following relationships are defined for Type Graphs (Table 8):

- `IS_SUBTYPE_EXTENDS`: relates a type definition to its direct supertypes.
- `IS_SUBTYPE_IMPLEMENTS`: relates a class or enum definition to its direct super-interfaces.
- `ITS_TYPE_IS`: relates expressions, and variable and method/constructor definitions to their type.
- `INHERITS_FIELD`: relates type definitions to their (directly or indirectly) inherited fields, if any.
- `INHERITS_METHOD`: relates type definitions to their (directly or indirectly) inherited methods, provided that they are not overridden.
- `OVERRIDES`: relates a method definition to the overridden method definition, if any.
- `ELEMENT_TYPE`: relates an array type to the type of its elements.
- `RETURN_TYPE`: relates a callable type to its return type.
- `PARAM_TYPE`: relates a callable type to its parameter types, if any.
- `THROWS_TYPE`: connects a callable type to the exceptions in its `throws` clause, if any.
- `INSTANCE_ARG_TYPE`: relates a constructor type to the type to be instantiated.
- `UPPER_BOUND_TYPE`: given `<` T_1 `extends` T_2 `>`, this relationship connects T_1 to T_2.
- `LOWER_BOUND_TYPE`: given `<? super` T `>`, this relationship connects the type that the compiler instantiates for `?` to T.
- `WILDCARD_EXTENDS_BOUND`: relates a wildcard to the type included in its `extends` clause, if any (e.g., `?` `extends` *Type*).
- `WILCARD_SUPER_BOUND`: relates a wildcard to the type included in its `super` clause, if any (e.g., `?` `super` *Type*).

The following properties are defined (Table 9):

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2987631, IEEE Access

TABLE 5. Relationships defined for CFGs.

Relationship	Domain	Range	Cardinality
CFG_ENTRIES	CALLABLE_DEF	CFG_ENTRY	0..1
CFG_END_OF	CFG_NORMAL_END ∪ CFG_EXCEPTIONAL_END	CALLABLE_DEF	1
CFG_FINALLY_TO_LAST_STMT	FINALLY_BLOCK	CFG_LAST_STATEMENT_IN_FINALLY	1
CFG_NEXT_STATEMENT	STATEMENT	CFG_NODE ∪ STATEMENT	0..1
CFG_NEXT_STATEMENT_IF_TRUE	ASSERT_STATEMENT ∪ DO_WHILE_LOOP ∪ FOR_LOOP ∪ IF_STATEMENT ∪ WHILE_LOOP	CFG_NODE ∪ STATEMENT	1
CFG_NEXT_STATEMENT_IF_FALSE	DO_WHILE_LOOP ∪ FOR_LOOP ∪ IF_STATEMENT ∪ WHILE_LOOP	CFG_NODE ∪ STATEMENT	1
CFG_FOR_EACH_HAS_NEXT	FOR_EACH_LOOP	STATEMENT	1
CFG_FOR_EACH_NO_MORE_ELEMENTS	FOR_EACH_LOOP	CFG_NODE ∪ STATEMENT	1
CFG_SWITCH_CASE_IS_EQUAL_TO	SWITCH_STATEMENT	CFG_NODE ∪ STATEMENT	0..*
CFG_SWITCH_DEFAULT_CASE	SWITCH_STATEMENT	CFG_NODE ∪ STATEMENT	0..1
CFG_AFTER_FINALLY_PREVIOUS_BREAK	LAST_STATEMENT_IN_FINALLY	CFG_NODE ∪ STATEMENT	0..1
CFG_AFTER_FINALLY_PREVIOUS_CONTINUE	LAST_STATEMENT_IN_FINALLY	STATEMENT	0..1
CFG_NO_EXCEPTION	LAST_STATEMENT_IN_FINALLY	CFG_NODE ∪ STATEMENT	1
CFG_IF_THERE_IS_UNCAUGHT_EXCEPTION	CATCH_BLOCK ∪ LAST_STATEMENT_IN_FINALLY	EXCEPTIONAL_END ∪ CATCH_BLOCK ∪ FINALLY_BLOCK ∪ LOCAL_VAR_DEF	0..1
CFG_CAUGHT_EXCEPTION	CATCH_BLOCK	LOCAL_VAR_DEF	0..1
CFG_MAY_THROW	STATEMENT	EXCEPTIONAL_END ∪ CATCH_BLOCK ∪ FINALLY_BLOCK ∪ LOCAL_VAR_DEF	0..1
CFG_THROWS	THROW_STATEMENT	EXCEPTIONAL_END ∪ CATCH_BLOCK ∪ FINALLY_BLOCK ∪ LOCAL_VAR_DEF	1

TABLE 6. Properties defined for CFGs.

Property	Type	Domain	Value-Type	Cardinality
mustBeExecuted	Node	STATEMENT	Boolean	1
exceptionType	Edge	CFG_THROWS ∪ CFG_MAY_THROW ∪ CFG_CAUGHT_EXCEPTION ∪ CFG_IF_THERE_IS_UNCAUGHT_EXCEPTION	String	1
methodName	Edge	CFG_MAY_THROW	String	0..1
label	Edge	AFTER_FINALLY_PREVIOUS_CONTINUE ∪ AFTER_FINALLY_PREVIOUS_BREAK	String	0..1
caseValue	Edge	CFG_SWITCH_CASE_IS_EQUAL_TO	String	1
caseIndex	Edge	CFG_SWITCH_CASE_IS_EQUAL_TO ∪ CFG_SWITCH_DEFAULT_CASE	Integer	1

TABLE 7. Relationships defined for Call Graphs.

Relationship	Domain	Range	Cardinality
CALLS	CALLABLE_DEF	CALL	0..*
HAS_DEF	CALL	CALLABLE_DEF	1
REFERS_TO	CALL	CALLABLE_DEF	0..1
MAY_REFER_TO	CALL	CALLABLE_DEF	0..*

- `actualType`: string representing the type of an expression, callable or variable definition.
- `typeKind`: string representing a type generalization (Table 9).
- `typeBoundKind`: string describing the kind of bound of a wildcard type (Table 9).

F. PROGRAM DEPENDENCY GRAPH

The following new nodes are defined for PDGs:

- `THIS_REF`: represents the implicit object (`this`) in each type definition.
- `INITIALIZATION`: represents the initialization of variable (attribute, parameter or local variable) definitions.

Relationships defined for PDGs (Table 10):

- `USED_BY`: relates a variable (field, parameter or local variable) definition to the expressions where the variable is read, if any.
- `MODIFIED_BY`: relates a variable definition to the expressions in which its value is modified.
- `STATE_MODIFIED_BY`: relates a variable definition or the implicit object (`this`) to the expressions or callable definitions where its state is certainly mutated, if any.
- `STATE_MAY_BE_MODIFIED_BY`: relates a variable definition or the implicit object (`this`) to the invocations or callable definitions where its state may be modified.
- `HAS_THIS_REFERENCE`: relates any type definition to the implicit object reference (`this`).

The property `isOwnAccess` is defined for the first four PDG relationships (0..1 cardinality and Boolean value-type). It indicates whether an expression accesses a field of the implicit object (`this`).

G. CLASS DEPENDENCY GRAPH

For CDGs, we define two relationships:

- `USES_TYPE_DEF`: connects two type definitions (declared in the project or not), representing that the source node depends on the target one. Therefore, its domain and range are `TYPE_DEFINITION`; its cardinality is 0..*.
- `HAS_INNER_TYPE_DEF`: relates a compilation unit to the inner types defined inside it. Its domain, range and cardinality are, respectively, `COMPILATION_UNIT`, `TYPE_DEFINITION` and 0..*.

TABLE 8. Relationships defined for Type Graphs.

Relationship	Domain	Range	Cardinality
IS_SUBTYPE_EXTENDS	TYPE_DEFINITION	TYPE_DEFINITION	0..*
IS_SUBTYPE_IMPLEMENTS	CLASS_DEF OR ENUM_DEF	INTERFACE_DEF	0..*
ITS_TYPE_IS	CALLABLE_DEF ∪ EXPRESSION ∪ VARIABLE_DEF	TYPE_NODE	1
INHERITS_FIELD	TYPE_DEFINITION	ATTR_DEF	0..*
INHERITS_METHOD	TYPE_DEFINITION	METHOD_DEF	0..*
OVERRIDES	METHOD_DEF	METHOD_DEF	0..1
ELEMENT_TYPE	ARRAY_TYPE	TYPE	1
RETURN_TYPE	CALLABLE_TYPE	TYPE	1
PARAM_TYPE	CALLABLE_TYPE	TYPE	0..*
THROWS_TYPE	CALLABLE_TYPE	TYPE	0..*
INSTANCE_ARG_TYPE	CALLABLE_TYPE	TYPE	0..1
UPPER_BOUND_TYPE	TYPE_VAR	TYPE	1
LOWER_BOUND_TYPE	TYPE_VAR	TYPE	1
WILDCARD_EXTENDS_BOUND	WILCARD_TYPE	TYPE	0..1
WILCARD_SUPER_BOUND	WILCARD_TYPE	TYPE	0..1

TABLE 9. Properties defined for Type Graphs.

Property	Type	Domain	Value-Type	Cardinality
actualType	Node	EXPRESSION ∪ CALLABLE_DEF ∪ VARIABLE_DEF	String	1
typeKind	Node	EXPRESSION ∪ CALLABLE_DEF ∪ VARIABLE_DEF	{ ARRAY, BOOLEAN, BYTE, CHAR, DECLARED, DOUBLE, EXECUTABLE, FLOAT, INT, INTERSECTION, LONG, NULL, PACKAGE, SHORT, TYPE_VAR, VOID, UNION, WILDCARD }	1
typeBoundKind	Node	WILCARD_TYPE	{ SUPER_WILDCARD, EXTENDS_WILDCARD, UNBOUNDED_WILDCARD }	1

TABLE 10. Relationships defined for PDGs.

Relationship	Domain	Range	Cardinality
USED_BY	VARIABLE_DEF	IDENTIFIER ∪ MEMBER_SELECTION	0..*
MODIFIED_BY	VARIABLE_DEF	ASSIGNMENT ∪ COMPOUND_ASSIGNMENT ∪ UNARY_OPERATION	0..*
STATE_MODIFIED_BY	VARIABLE_DEF ∪ THIS_REF	ASSIGNMENT ∪ COMPOUND_ASSIGNMENT ∪ UNARY_OPERATION ∪ CALL ∪ CALLABLE_DEFINITION	0..*
STATE_MAY_BE_MODIFIED_BY	VARIABLE_DEF ∪ THIS_REF	CALL ∪ CALLABLE_DEFINITION	0..*
HAS_THIS_REFERENCE	TYPE_DEFINITION	THIS_REF	0..1

H. PACKAGE GRAPH

Two nodes are added for Package Graphs:

- PACKAGE: represents any package declaration defined or used in the program.
- PROGRAM: models the whole program, representing the graph root.

What follows are the Package Graph relationships defined (details in Table 11):

- PROGRAM_DECLARES_PACKAGE: relates a program to the packages defined in it.
- PACKAGE_HAS_COMPILATION_UNIT: relates a package to the compilation units it contains.
- DEPENDS_ON_PACKAGE: relates a package to the packages it depends on, if any; target packages must be defined in the source code.
- DEPENDS_ON_NON_DECLARED_PACKAGE: relates a package to the packages it depends on, if any; target packages are not defined in the source code.

Finally, the following two properties are included in Package Graphs:

- ID: node property defined for PROGRAM. It is a unique identifier for each program. Its value-type is string and has cardinality of one.
- timestamp: a property of the PROGRAM node indicating when the program was inserted in the database. Its value-type is date and has cardinality of one.

REFERENCES

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

[2] R.-G. Urma and A. Mycroft, "Programming language evolution via source code query languages," in Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools, ser. PLATEAU '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 35–38.

[3] F. Nielson, H. R. Nielson, and C. Hankin, Principles of Program Analysis. Springer Publishing Company, Incorporated, 2010.

[4] F. Ortin, J. Escalada, and O. Rodriguez-Prieto, "Big Code: new opportunities for improving software construction," Journal of Software, vol. 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 Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 457–466.

TABLE 11. Relationships defined for Package Graphs.

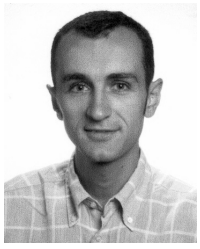
Relationship	Domain	Range	Cardinality
PROGRAM_DECLARES_PACKAGE	PROGRAM	PACKAGE	1..*
PACKAGE_HAS_COMPILATION_UNIT	PACKAGE	COMPILATION_UNIT	1..*
DEPENDS_ON_PACKAGE	PACKAGE	PACKAGE	0..*
DEPENDS_ON_NON_DECLARED_PACKAGE	PACKAGE	PACKAGE	0..*

[6] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," IEEE Software, vol. 25, no. 5, pp. 22–29, 2008.
[7] R. Ivanov, "Checkstyle," https://checkstyle.sourceforge.io, 2020.
[8] Coverity, "Coverity scan static analysis," https://scan.coverity.com, 2020.
[9] R. G. Urma and A. Mycroft, "Source-code queries with graph databases—with application to programming language usage and evolution," Science of Computer Programming, vol. 97, pp. 127–134, Jan. 2015.
[10] Google, "BigQuery," https://cloud.google.com/bigquery, 2020.
[11] Raoul-Gabriel Urma, "Wiggle," https://github.com/raoulDoc/WiggleIndexer, 2020.
[12] V. Raychev, M. Vechev, and A. Krause, "Predicting Program Properties from "Big Code"," in Proceedings of the 42^{nd} Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '15, New York, NY, USA, 2015, pp. 111–124.
[13] Defense Advanced Research Projects Agency, "MUSE Envisions Mining "Big Code" to Improve Software Reliability and Construction," http://www.darpa.mil/news-events/2014-03-06a, 2014.
[14] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-Based Statistical Translation of Programming Languages," in Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, ser. Onward! 2014. New York, NY, USA: ACM, 2014, pp. 173–184.
[15] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized Vulnerability Extrapolation Using Abstract Syntax Trees," in Proceedings of the 28th Annual Computer Security Applications Conference, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 359–368.
[16] J. Escalada, F. Ortin, and T. Scully, "An Efficient Platform for the Automatic Extraction of Patterns in Native Code," Scientific Programming, vol. 2017, pp. 1–16, 2017.
[17] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective." in ACM Southeast Regional Conference, H. C. Cunningham, P. Ruth, and N. A. Kraft, Eds. ACM, 2010, p. 42.
[18] CMU SEI, "Carnegie Mellon University, Software Engineering Institute, Java coding guidelines," https://wiki.sei.cmu.edu/confluence/display/java/Java+Coding+Guidelines, 2020.
[19] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in Proceedings of the 2018 International Conference on Management of Data, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1433–1445.
[20] R. G. Urma, "Programming language evolution," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-902, Feb. 2017.
[21] Semmle, "CodeQL," https://semmle.com/codeql, 2020.
[22] S. S. Huang, T. J. Green, and B. T. Loo, "Datalog and emerging applications: An interactive tutorial," in Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1213–1216.
[23] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik, "The object-oriented database system manifesto," in Deductive and Object-Oriented Databases. Elsevier, 1990, pp. 223–240.
[24] F. Dietze, J. Karoff, A. Calero Valdez, M. Ziefle, C. Greven, and U. Schroeder, "An Open-Source Object-Graph-Mapping Framework for Neo4j and Scala: Renesca," in International Conference on Availability, Reliability, and Security (CD-ARES), ser. Availability, Reliability, and Security in Information Systems, F. Buccafurri, A. Holzinger, P. Kieseberg, A. M. Tjoa, and E. Weippl, Eds., vol. LNCS-9817. Salzburg, Austria: Springer International Publishing, 2016, pp. 204–218.
[25] N. Hawes, B. Barham, and C. Cifuentes, "Frappé: Querying the linux kernel dependency graph," in Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015, 2015, pp. 4:1–4:6.
[26] O. Goonetilleke, D. Meibusch, and B. Barham, "Graph data management of evolving dependency graphs for multi-versioned codebases," in 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017, 2017, pp. 574–583.
[27] T. Zhang, M. Pan, J. Zhao, Y. Yu, and X. Li, "An open framework for semantic code queries on heterogeneous repositories," in 2015 International Symposium on Theoretical Aspects of Software Engineering. IEEE, Sep. 2015, pp. 39–46.
[28] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow, "TXL: A rapid prototyping system for programming language dialects," Computer Languages, vol. 16, no. 1, pp. 97–107, Jan. 1991.
[29] P. Membrey, The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing (Expert's Voice in Open Source). Apress, oct 2010.
[30] T. Zhang, X. Zheng, Y. Zhang, J. Zhao, and X. Li, "A declarative approach for Java code instrumentation," Software Quality Journal, vol. 23, no. 1, pp. 143–170, Sep. 2013.
[31] G. Erich, H. Richard, J. Ralph, and V. John, Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional Computing Series, 1995.
[32] G. A. Campbell and P. P. Papapetrou, SonarQube in Action. Manning Publications, nov 2013.
[33] M. Fowler, Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, jul 1999.
[34] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," Electronic Notes in Theoretical Computer Science, vol. 217, pp. 5–21, 2008.
[35] PMD, "PMD Source Code Analyzer Project," https://pmd.github.io, 2020.
[36] J. Bloch, Effective Java (The Java Series), 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.
[37] L. Tahvildari and A. Singh, "Categorization of object-oriented software metrics," in 2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era (Cat. No.00TH8492). IEEE, 2000, pp. 235–239.
[38] F. Ortin, O. Rodriguez-Prieto, N. Pascual, and M. Garcia, "Heterogeneous tree structure classification to label java programmers according to their expertise level," Future Generation Computer Systems, vol. 105, pp. 380–394, Apr. 2020.
[39] M. A. Rodriguez, "The Gremlin graph traversal machine and language (invited talk)," in Proceedings of the 15th Symposium on Database Programming Languages, ser. DBPL 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–10.
[40] Neo4j, "The Neo4j traversal framework," https://neo4j.com/docs/java-reference/current/tutorial-traversal, 2020.
[41] F. Ortin, D. Zapico, and J. M. Cueva, "Design Patterns for Teaching Type Checking in a Compiler Construction Course," IEEE Transactions on Education, vol. 50, no. 3, pp. 273–283, 2007.
[42] CUP research group, "GitHub Java corpus," http://groups.inf.ed.ac.uk/cup/javaGithub, 2020.
[43] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 281–293.
[44] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–265.

[45] M. Goeminne and T. Mens, "Towards a survival analysis of database framework usage in Java projects," in Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), ser. ICSME '15. USA: IEEE Computer Society, 2015, p. 551–555.
[46] CMU CERT, "Carnegie Mellon University, CERT Division, Software Engineering Institute," https://www.sei.cmu.edu/about/divisions/cert, 2020.
[47] R. C. Seacord and J. A. Rafail, "Secure coding standards," in Proceedings of the Static Analysis Summit, NIST Special Publication, 2006, pp. 13–17.
[48] Oracle, "Java Platform Standard Edition 7 Documentation," https://docs.oracle.com/javase/7/docs, 2013.
[49] K. Havelund and A. Niessner, "JPL Coding Standard, Version 1.1," https://www.havelund.com/Publications/JavaCodingStandard.pdf, 2010.
[50] Oracle, "Java Platform, Standard Edition API Specification," https://docs.oracle.com/javase/8/docs/api/index.html, 2020.
[51] K. Knoernschild, Java Design: Objects, UML, and Process. Indianapolis, IN, USA: Addison-Wesley Professional, 2002.
[52] A. Sterbenz and C. Lai, "Secure coding antipatterns: Avoiding vulnerabilities," in JavaOne Conference, 2006.
[53] Oracle, "The Java Tutorials," https://docs.oracle.com/javase/tutorial, 2020.
[54] O. Rodriguez-Prieto and F. Ortin, "An efficient and scalable platform for java source code analysis using overlaid graph representations (support material website)," http://www.reflection.uniovi.es/bigcode/download/2020/ieee-access, 2019.
[55] A. Vukotic, N. Watt, T. Abedrabbo, D. Fox, and J. Partner, Neo4j in Action, 1st ed. USA: Manning Publications Co., 2014.
[56] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "QL: Object-oriented Queries on Relational Data," in 30th European Conference on Object-Oriented Programming (ECOOP 2016), ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 2:1–2:25.
[57] Semmle, "Variant Analysis," https://semmle.com/variant-analysis, 2020.
[58] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, ser. OOPSLA. New York, NY, USA: ACM, 2007, pp. 57–76.
[59] D. J. Lilja, Measuring computer performance: a practitioner's guide. Cambridge University Press, 2005.
[60] F. Ortin, M. A. Labrador, and J. M. Redondo, "A hybrid class- and prototype-based object model to support language-neutral structural intercession," Information and Software Technology, vol. 44, no. 1, pp. 199–219, feb 2014.
[61] Microsoft, "Windows management instrumentation," http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582(v=vs.85).aspx, 2015.
[62] GQL, "Graph Query Language, GQL Standard," https://www.gqlstandards.org, 2020.
[63] Semmle, "LGTM, continuous security analysis," https://lgtm.com, 2020.
[64] F. Ortin, M. A. Labrador, and J. M. Redondo, "A hybrid class- and prototype-based object model to support language-neutral structural intercession," Information and Software Technology, vol. 56, no. 2, p. 199–219, Feb. 2014.
[65] J. Bellamy-McIntyre, "Modeling and querying versioned source code in rdf," in The Semantic Web: ESWC 2018 Satellite Events, A. Gangemi, A. L. Gentile, A. G. Nuzzolese, S. Rudolph, M. Maleshkova, H. Paulheim, J. Z. Pan, and M. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 251–261.
[66] O. Rodriguez-Prieto and F. Ortin, "Graph Representations used in the design of ProgQuery," University of Oviedo, Computational Reflection research group, Tech. Rep., Jan. 2020. [Online]. Available: http://www.reflection.uniovi.es/bigcode/download/2020/ieee-access/tr.pdf

OSCAR RODRIGUEZ-PRIETO is a full-time PhD student at the Computer Science Department of the University of Oviedo, Spain. He received his BSc degree in Software Engineering in 2014. In 2015, he was awarded a MSc in Artificial Intelligence from the Spanish National Distance Education University (UNED). His PhD thesis aims at using the idea of big code to improve software reliability and construction.

ALAN MYCROFT is Professor of Computing in the Department of Computer Science and Technology in the University of Cambridge where he has worked since 1984. After a degree in Mathematics at Cambridge, Mycroft took his PhD at the University of Edinburgh. He has published around 100 papers in topics centred on "programming languages, their static analysis and compilation". He has taken sabbaticals in research laboratories (AT&T and Intel), co-authored the 'Norcroft C compiler' for ARM and was a co-founder of the Raspberry Pi Foundation.



FRANCISCO ORTIN is a Full Professor of the Computer Science Department at the University of Oviedo, Spain. He also works as an Adjunct Lecturer for the Cork Institute of Technology (Ireland). He is the head of the Computational Reflection research group (http://www.reflection.uniovi.es). He received his BSc in Computer Science in 1994, and his MSc in Computer Engineering in 1996. In 2002, he was awarded his PhD degree. He has been the principal investigator of different research projects funded by Microsoft Research and the Spanish Department of Science and Innovation. His main research interests include big code, dynamic languages, type systems, aspect-oriented programming, and runtime adaptable applications. Contact him at http://www.reflection.uniovi.es/ortin.