Infraestructura Big Code para Mejorar la Calidad en el Desarrollo de Software

Autor

Óscar Rodríguez Prieto

Director

Francisco Ortín Soler

Programa de Doctorado en Informática

Departamento de Informática

Universidad de Oviedo

Oviedo, España

Abril de 2020

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

RESUMEN DEL CONTENIDO DE TESIS DOCTORAL

1.- Título de la Tesis	
Español: Infraestructura Big Code para Mejorar la Calidad en el Desarrollo de Software	Inglés: Big Code Infrastructure for Building Tools to Improve Software Development

2.- Autor	
Nombre: Óscar Rodríguez Prieto	DNI:
Programa de Doctorado: Ingeniería Informática	
Órgano responsable: Departamento de Informática	

RESUMEN (en español)

En la última década, el uso de repositorios de código como GitHub, SourceForge o BitBucket se ha visto incrementado significativamente. Según los datos de GitHub, en noviembre de 2018 se había registrado 100 millones de repositorios; posteriormente, en 2019 se crearon 44 millones de nuevos repositorios. La información de estos repositorios de código masivos puede ser utilizada como datos para la construcción de herramientas, servicios y modelos orientados a mejorar el desarrollo software. La nueva área de investigación sobre este tema ha sido bautizada como "big code", debido a su paralelismo con el big data y al uso de código fuente (source code) como datos.

Aunque la gran mayoría de los programas se codifican como información textual, éstos contienen información sintáctica y semántica típicamente representada por medio de árboles y grafos. Las herramientas para mejorar el desarrollo software extraen dicha información y la utilizan para distintas tareas como el análisis estático de programas, la desofuscación de código, el cálculo de métricas software, la transformación de programas, la búsqueda avanzada de código y la comprobación de guías de estilo. No obstante, la mayoría de estas herramientas se diseñan *ad hoc* para resolver la tarea específica para la que fueron construidas. Por otra parte, suele existir un compromiso entre la escalabilidad de los sistemas y el nivel de detalle de la información que proporcionan sobre el código fuente: los que soportan programas de millones de líneas de código tan solo proveen un subconjunto reducido de la información.

En esta tesis doctoral se presenta ProgQuery, una infraestructura que permite a los usuarios la creación de sus propias herramientas orientadas a procesar la información sintáctica y semántica del código fuente. Se diseñaron 7 representaciones basadas en grafos que modelan distinta información sintáctica y semántica relativa a programas Java. Estas representaciones están superpuestas unas con otras, es decir, los nodos sintácticos y semánticos de los distintos grafos se interconectan entre s´ı, facilitando la combinación de distintos tipos de información.

Se modifico´ el compilador de Java para procesar dichas representaciones y almacenarlas en una base de datos orientada a grafos Neo4j. Al emplear la misma estructura de datos en el sistema de persistencia, se evitan los desajustes de impedancia derivados de utilizar bases de datos relacionales. Algunas ventajas de esta aproximación son el acceso directo a las entidades de los grafos, mayor

SR. PRESIDENTE DE LA COMISIÓN ACADÉMICA DEL PROGRAMA DE DOCTORADO

rendimiento que el modelo relacional y la posibilidad de utilizar diversos mecanismos para recuperar la información de la base de datos.

En esta tesis utilizamos la infraestructura propuesta en dos escenarios. Primero, haciendo uso de Cypher, un lenguaje declarativo de consulta para grafos, se implementaron en ProgQuery 13 análisis esta´ticos publicados por el equipo CERT del Instituto de Ingeniería del Software de la Universidad de Carnegie Mellon. Estos análisis se expresan en forma de consultas que detectan errores comunes de programación en Java, no detectados por los compiladores. El segundo escenario consiste en la extracción de información de programas para clasificar código fuente, aplicando aprendizaje automático supervisado. La herramienta desarrollada es capaz de determinar si un programador es experto o novato con una precisión media del 99,6 %.

Finalmente, se llevó a cabo la evaluación de ProgQuery, comparando su rendimiento con el de los sistemas relacionados existentes. Nuestro sistema es más eficiente que los demás en cuanto al tiempo de análisis y, además, es más escalable al tamaño de los programas y a la complejidad de los análisis. ProgQuery es capaz de analizar los programas Java de mayor tamaño existentes en los repositorios (decenas de millones de líneas de código) en decenas de segundos, mientras el resto de los sistemas evaluados muestran errores por falta de memoria. Adicionalmente, la cantidad de código empleada para codificar las consultas indica que ProgQuery es más expresivo que las otras aproximaciones. Sin embargo, la información adicional almacenada por nuestro sistema hace que el tiempo de inserción de los programas y el tamaño de la base de datos aumente, pero estos incrementos son significativamente menores que las mejoras de rendimiento obtenidas.

RESUMEN (en inglés)

The use of source code repositories such as GitHub, SourceForge and Bitbucket has significantly risen in the last decade. According to GitHub, on November 2018 this code hosting platform reached 100 million repositories; on 2019, 44 million new repositories were created. These massive codebases could be used as data to create programming tools, services and models to improve software development. The research field focused on this topic has been termed “big code”, due to its similarity with big data and the usage of source code.

Most programs are commonly written as textual information. However, they enclose syntactic and semantic information that is usually represented as trees and graphs. This information is retrieved by software tools, and used for many different purposes such as static program analysis, code deobfuscation, software metrics computation, program transformation, advanced code search, and coding guideline checking. However, most of such tools are designed ad hoc for the particular purpose they are aimed at. Moreover, there is usually a trade-off between scalability and the source-code detail provided: those tools that scale to million-line programs provide a reduced subset of program information.

This dissertation proposes an infrastructure, called ProgQuery, to allow users to create their own tools to process the syntactic and semantic information of source

code. Seven graph representations are designed to model different syntactic and semantic information of Java programs. Such representations are overlaid, meaning that syntactic and semantic nodes of the different graphs are interconnected to allow combining different kinds of information.

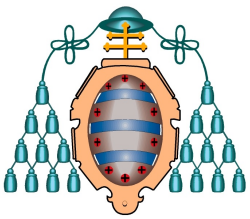
We modify the Java compiler to compute the syntactic and semantic representations, and store them in a Neo4j graph database. By using the same data structure in the persistence system, we avoid the impedance mismatch caused by the utilization of a relational database. Main benefits are the direct access to graph entities in the persistence store, better performance than the relational model, and multiple mechanisms to retrieve the information from the database.

In this dissertation, we use the proposed system for two case scenarios. First, we use the Cypher declarative graph query language to implement in ProgQuery 13 static analyses published by the CERT division of the Software Engineering Institute at Carnegie Mellon University. These analyses are expressed as queries, and prompt common Java

mistakes not detected by the compiler. The second case scenario is the extraction of program information to classify source code using supervised machine learning. The system is able to label expert and novice programmers with an average accuracy of 99.6%.

We evaluate ProgQuery and compare it to the related systems. Our system outperforms the other systems in analysis time, and scales better to program size and analysis complexity. ProgQuery analyzes huge Java programs in tens of seconds, whereas the other alternatives show memory errors. Moreover, the amount of source code required to implement the analyses shows that ProgQuery is more expressive than the other approaches. The additional information stored by ProgQuery increases insertion time of programs and database size, but these increases are significantly lower than the analysis performance gains obtained.

Big Code Infrastructure for Building Tools to Improve Software Development



Óscar Rodríguez Prieto

PhD Supervisor

Prof. Francisco Ortín Soler

Department of Computer Science

University of Oviedo

A thesis submitted for the degree of

Doctor of Philosophy

Oviedo, Spain

February 2020

Acknowledgements

This work has been partially funded by the Spanish Department of Science and Technology, under the National Program for Research, Development and Innovation (project RTI2018-099235-B-I00). We have also received funds from the University of Oviedo, through its support to official research groups (GR-2011-0040).

I was awarded an FPU grant by the Spanish Department of Science and Technology (grant number FPU15/05261). The objective of these grants is to support graduate students wishing to pursue a PhD degree. A PhD dissertation is proposed to be undertaken by the applicant, within a research group and supervised by a tenure researcher.

Part of the research discussed in this dissertation has also been funded by the European Union, through the European Regional Development Funds (ERDF); and the Principality of Asturias, through its Science, Innovation Plan (grant GRUPIN14-100).

The research work in this dissertation was partially done during my research stay at the Computer Laboratory of the University of Cambridge (UK), under the supervision of Professor Alan Mycroft. I thank all the people of the Computer Laboratory for their warm welcome and their generous hospitality.

Abstract

The use of source code repositories such as GitHub, SourceForge and Bitbucket has significantly risen in the last decade. According to GitHub, on November 2018 this code hosting platform reached 100 million repositories; on 2019, 44 million new repositories were created. These massive codebases could be used as data to create programming tools, services and models to improve software development. The research field focused on this topic has been termed "big code", due to its similarity with big data and the usage of source code.

Most programs are commonly written as textual information. However, they enclose syntactic and semantic information that is usually represented as trees and graphs. This information is retrieved by software tools, and used for many different purposes such as static program analysis, code deobfuscation, software metrics computation, program transformation, advanced code search, and coding guideline checking. However, most of such tools are designed *ad hoc* for the particular purpose they are aimed at. Moreover, there is usually a trade-off between scalability and the source-code detail provided: those tools that scale to million-line programs provide a reduced subset of program information.

This dissertation proposes an infrastructure, called ProgQuery, to allow users to create their own tools to process the syntactic and semantic information of source code. Seven graph representations are designed to model different syntactic and semantic information of Java programs. Such representations are overlaid, meaning that syntactic and semantic nodes of the different graphs are interconnected to allow combining different kinds of information.

We modify the Java compiler to compute the syntactic and semantic representations, and store them in a Neo4j graph database. By using the same data structure in the persistence system, we avoid the impedance mismatch caused by the utilization of a relational database. Main benefits are the direct access to graph entities in the persistence store, better performance than the relational model, and multiple mechanisms to retrieve the information from the database.

In this dissertation, we use the proposed system for two case scenarios. First, we use the Cypher declarative graph query language to implement in ProgQuery 13 static analyses published by the CERT division of the Software Engineering Institute at Carnegie Mellon University. These analyses are expressed as queries, and prompt common Java

mistakes not detected by the compiler. The second case scenario is the extraction of program information to classify source code using supervised machine learning. The system is able to label expert and novice programmers with an average accuracy of 99.6%.

We evaluate ProgQuery and compare it to the related systems. Our system outperforms the other systems in analysis time, and scales better to program size and analysis complexity. ProgQuery analyzes huge Java programs in tens of seconds, whereas the other alternatives show memory errors. Moreover, the amount of source code required to implement the analyses shows that ProgQuery is more expressive than the other approaches. The additional information stored by ProgQuery increases insertion time of programs and database size, but these increases are significantly lower than the analysis performance gains obtained.

Keywords

Big code, program representation, static program analysis, graph database, coding guidelines, declarative query language, machine learning, syntax patterns, abstract syntax trees, decision trees, program expertise, Cypher, Java, Neo4j

Resumen

En la última década, el uso de repositorios de código como GitHub, SourceForge o BitBucket se ha visto incrementado significativamente. Según los datos de GitHub, en noviembre de 2018 se había registrado 100 millones de repositorios; posteriormente, en 2019 se crearon 44 millones de nuevos repositorios. La información de estos repositorios de código masivos puede ser utilizada como datos para la construcción de herramientas, servicios y modelos orientados a mejorar el desarrollo software. La nueva área de investigación sobre este tema ha sido bautizada como "big code", debido a su paralelismo con el *big data* y al uso de código fuente (*source code*) como datos.

Aunque la gran mayoría de los programas se codifican como información textual, éstos contienen información sintáctica y semántica típicamente representada por medio de árboles y grafos. Las herramientas para mejorar el desarrollo software extraen dicha información y la utilizan para distintas tareas como el análisis estático de programas, la desofuscación de código, el cálculo de métricas software, la transformación de programas, la búsqueda avanzada de código y la comprobación de guías de estilo. No obstante, la mayoría de estas herramientas se diseñan *ad hoc* para resolver la tarea específica para la que fueron construídas. Por otra parte, suele existir un compromiso entre la escalabilidad de los sistemas y el nivel de detalle de la información que proporcionan sobre el código fuente: los que soportan programas de millones de líneas de código tan solo proveen un subconjunto reducido de la información.

En esta tesis doctoral se presenta ProgQuery, una infraestructura que permite a los usuarios la creación de sus propias herramientas orientadas a procesar la información sintáctica y semántica del código fuente. Se diseñaron 7 representaciones basadas en grafos que modelan distinta información sintáctica y semántica relativa a programas Java. Estas representaciones están superpuestas unas con otras, es decir, los nodos sintácticos y semánticos de los distintos grafos se interconectan entre sí, facilitando la combinación de distintos tipos de información.

Se modificó el compilador de Java para procesar dichas representaciones y almacenarlas en una base de datos orientada a grafos Neo4j. Al emplear la misma estructura de datos en el sistema de persistencia, se evitan los desajustes de impedancia derivados de utilizar bases de datos relacionales. Algunas ventajas de esta aproximación son el acceso directo a las entidades de los grafos, mayor rendimiento que el modelo relacional y la posibilidad de utilizar diversos mecanismos

para recuperar la información de la base de datos.

En esta tesis utilizamos la infraestructura propuesta en dos escenarios. Primero, haciendo uso de Cypher, un lenguaje declarativo de consulta para grafos, se implementaron en ProgQuery 13 análisis estáticos publicados por el equipo CERT del Instituto de Ingeniería del Software de la Universidad de Carnegie Mellon. Estos análisis se expresan en forma de consultas que detectan errores comunes de programación en Java, no detectados por los compiladores. El segundo escenario consiste en la extracción de información de programas para clasificar código fuente, aplicando aprendizaje automático supervisado. La herramienta desarrollada es capaz de determinar si un programador es experto o novato con una precisión media del 99,6 %.

Finalmente, se llevó a cabo la evaluación de ProgQuery, comparando su rendimiento con el de los sistemas relacionados existentes. Nuestro sistema es más eficiente que los demás en cuanto al tiempo de análisis y, además, es más escalable al tamaño de los programas y a la complejidad de los análisis. ProgQuery es capaz de analizar los programas Java de mayor tamaño existentes en los repositorios (decenas de millones de líneas de código) en decenas de segundos, mientras el resto de sistemas evaluados muestran errores por falta de memoria. Adicionalmente, la cantidad de código empleada para codificar las consultas indica que ProgQuery es más expresivo que las otras aproximaciones. Sin embargo, la información adicional almacenada por nuestro sistema hace que el tiempo de inserción de los programas y el tamaño de la base de datos aumente, pero estos incrementos son significativamente menores que las mejoras de rendimiento obtenidas.

Palabras Clave

Big code, representación de programas, análisis estático de programas, guías de codificación, lenguaje de consulta declarativo, aprendizaje automático, patrones sintácticos, árboles de sintaxis abstracta, árboles de decisión, experiencia de programación, Cypher, Java, Neo4j

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

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 models 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 `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 5.4.1.3).

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.

In this dissertation, we propose an efficient and scalable infrastructure, called ProgQuery, that allows the user to extract and utilize advanced semantic and syntactic properties of Java source code in a declarative and expressive way. Our work is based on the idea that programs can be represented with overlaid 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. Information of program representations can be extracted to create predictive models to improve software development such as source code classifiers [17].

1.2 Motivating examples

Before detailing the contribution of the proposed infrastructure, we describe two motivating examples where ProgQuery has been used. These two case scenarios are detailed later in this document (Chapters 5 and 6).

1.2.1 Static program analysis

We first illustrate how our system can be used for static program analysis. We describe how we implemented the OBJ50-J Java analysis/recomendation 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 1.1), 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 by the reference (i.e., believing the compiler prompts an error in line 26 of Figure 1.1). Therefore, many programmers improperly use `final` references, when they actually want to avoid mutability of the referenced object.

Figure 1.2 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 1.2 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 1 (Figure 1.2) matches all 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 1.1 (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 1.1). 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 2 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 1.2 gets the class (`mutatorEnclClass`) and Java file (`mutatorCU`) where the `mutatorMethod`s 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 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 5, 6 and 7 in Figure 1.2 are aimed at building and returning the warning message. Line 5 uses the common `reduce` higher order function to build a single string indicating all the code locations where the `final` variable is mutated. Line 6

[1] Line 26 in Figure 1.1 modifies the state of the `points` reference belonging to the implicit object (i.e., the object pointed by `this`), so `mutation.isOwnAccess` in line 4 of Figure 1.2 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`).

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

```java
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 1.1: Example Java code.

simply sets to `variableCU` the file where the final variable is defined, and line 7 returns the warning message. For the Java program in Figure 1.1, 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'.*

With this motivating example, we show 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

```
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 1.2: Cypher code implementing the OBJ50-J CERT CMU Java recommendation.

and scalability than the existing systems, with no additional memory consumption (see Section 5.4).

Other static analyses are detailed and evaluated in Chapter 5.

1.2.2 Source code classification

One of the challenges of big code is to classify and predict properties of source code using machine learning and probabilistic reasoning [4]. In this second motivating example, we use the syntactic information gathered by ProgQuery to classify and score the level of programming expertise of Java developers [17]. That classifier could be used for different purposes such as improving the messages given to programmers depending on their expertise level, documenting recurrent idioms used by experts and beginners, checking student's progress in a programming course, and building Intelligent Tutoring Systems (ITS) [20].

In order to build the classifier model, we took Java code from different sources and labeled them as either beginner or expert. For beginners, we gathered code from first year undergraduate students in a Software Engineering degree at the University of Oviedo. We took the code they wrote for the assignments in two year-1 programming courses in academic years 2017/18 and 2018/19. All the

code was 100% written by students from scratch. Overall, we collected 35,309 Java files from 3,884 programs.

For expert programmers, we took the source code of different public Java projects in GitHub. We selected active projects with the highest number of contributors: Chromium, LibreOffice, MySQL, OpenJDK and Amazon Web Services. These software products are implemented with 43,775 Java files in 137 programs (AWS comprises 133 different projects).

We extracted all the syntactic information detailed in Appendix A, using the Java compiler plug-in included in the ProgQuery infrastructure (Section 3.2). In this case scenario, we only used the syntactic information (not the semantic one), since we defined a feature engineering approach only valid for trees [17].

The classifiers consider different levels of syntax constructs, such as expressions, statements, methods, fields, types (classes, interfaces and enumerations) and whole programs. The obtained models are able to classify the expertise level of programmers with 78.1% accuracy for expressions, 88.3% for statements, 91.4% for fields, 95.2% for methods, 99.5% for types, and 99.6% for whole programs.

The system is also able to find recurrent idioms used by experts and beginners. For example, the following syntax pattern classifies a programmer as expert:

```
if enumeration_percentage(program)>0 and interface_percentage(program)>1
    and ∃ type1 . category(type1)==class and generic_types(type1)>0
    and ∃ type2 . category(type2)==class and extend_classes(type2)>0
        and implement_interfaces(type2)>1
    and ∃ method . number_statements(method)<=3
then expertise_level = expert
```

The previous pattern describes programs that contains enumeration and interface (more than 1%) types, implements no less than one generic type, at least one class extends another class and implements more than one interface, and one method has three or fewer statements.

Likewise, the following pattern to classify beginners was found:

```
if not(isOverride(method)) and numberOfAnnotations(method)==0
    and numberOfParameters(method)==0 and not(isFinal(method))
    and numberOfThrows(method)==0 and numberOfStatements(method)<=2
    and visibility(method)==public and numberOfGenericTypes(method)==0
    and namingConvention(method)==snake_case
    and ∃ statement . depth(statement)>=67
then expertise_level = beginner
```

This second pattern identifies non-`final` non-generic methods, with neither annotations nor parameters, which declare no exception thrown and are not overriding another method, use snake-case naming convention, and one statement has

a depth (distance from that statement and the furthest leaf node in the Abstract Syntax Tree (AST)) greater than 66.

Chapter 6 details how the classifiers were created. It also presents a model to score the level of expertise of programmers, and information about the syntax patterns extracted.

1.3 Contributions

These are the major contributions of this dissertation:

1. An infrastructure to provide syntactic and semantic information from Java source code (Chapter 3). Such information can be consulted declaratively, using different graph representations. These representations are overlaid, sharing common interconnected nodes that make it easy to combine syntactic and semantic program information.

2. Different graph structures comprising an ontology of syntactic and semantic representation of Java programs (Appendix A). Program information is represented with nodes (classes or concepts), relationships (relations) and properties (attributes or features).

3. An efficient and scalable storage (and inspection) of program representations (Chapters 3 and 5). Following the ontology designed, a Java compiler is modified to create graphs representing the input program and store them into a persistence system. No impedance mismatch exists, since graph abstractions are maintained in the database, avoiding the translation to tables. The resulting system provides significant lower analysis times than related systems, and scales better to program size and analysis complexity.

4. Utilization of the proposed infrastructure to implement different well-known static program analyses (Chapter 5). We take some analyses from CERT division of the Software Engineering Institute at Carnegie Mellon University [18], and implement them using ProgQuery. The analysis time, memory consumption, scalability for increasing program size and analysis complexity, and insertion time are measured and compared to related systems.

5. Web application prototype to automize and share static analysis of Java programs (Chapter 4). The ProgQuery infrastructure is offered as a web application and Web API (a set of web services) to allow the automatic insertion of programs, by specifying its GitHub identifier and using the Maven build automation tool. Existing analyses are provided. Users can create their own analyses, and share them with other users. Static analyses can be executed against single Java projects or a group of them.

6. Construction of a predictive model to classify Java code according to the programmer's expertise level (Chapter 6). Program information is obtained and translated into different datasets representing common Java constructs. This information is used in a feature learning process to build a classifier using supervised machine learning. The model is able to label expert and

novice programs with an average accuracy of 99.6%, only using syntactic information.

1.4 Structure of the document

This PhD dissertation is structured as follows. The next chapter describes related work, and the ProgQuery infrastructure is presented in Chapter 3. We describe the web prototype developed using the ProgQuery infrastructure in Chapter 4. Chapter 5 shows how to use ProgQuery to write real static program analyses, and uses those analyses to evaluate our system and compare it with related approaches. In Chapter 6, we show how ProgQuery can be used to extract program information to create a Java source-code classifier. Conclusions are presented in Chapter 7 and future work is discussed in Chapter 8.

Appendix A details the ontology designed to represent syntactic and semantic information of Java programs. Appendix B depicts the Cypher source code to implement in ProgQuery the static program analyses presented in Section 5.1. The features of the homogeneous datasets created with ProgQuery for the source-code classifier (Chapter 6) are presented in Appendix C. Finally, Appendix D enumerates the publications I wrote in the last four years.

Chapter 2

Related Work

This section describes the existing research works related to this PhD dissertation. We start describing the source code query tools (Section 2.1), which provide users with mechanisms to describe their own static analyses. Section 2.2 describe well-known static analysis tools that detect numerous errors in source code, but they do not allow users to write their own analyses. Next sections discuss the work related to source code classification with syntactic models: programmer classification (Section 2.3), clone detection (Section 2.4), other scenarios of syntactic classifiers (Section 2.5), and structured methods (Section 2.6).

2.1 Source code query tools

The closest work to the one presented in this dissertation is Wiggle, a prototype source-code querying system based on the graph data model [9]. It modifies the Java compiler to obtain ASTs of each program and store them in a Neo4j graph database. Wiggle proposes overlays as a mechanism to express queries as a mixture of syntactic and semantic information [21]. 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 5.4.4). 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++ [22]. 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. 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 [23]. Although QL provides graph abstractions to consult program representations, its storage in a relational database causes an impedance mismatch [24, 25]. Some evaluations have shown that graph databases perform better than relational ones when traversing graph structures [26].

Frappé is a C/C++ source code query tool that supports large-scale codebases [27]. A Neo4j graph database is chosen to gain query efficiency by avoiding repeated join operations, necessary in the relational model. 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. It has been used to perform queries against the Linux kernel (11.4 million lines of code), and to manage multiple source code versions [28].

Zhang *et al.* propose a source code query framework to support syntactic and semantic code queries across different object-oriented programming languages [29]. They handle language heterogeneity by transforming source code into a unified abstract syntax format, using TXL [30]. Program representations are stored in MangoDB, a Python wrapper for MongoDB [31]. 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 the JIns declarative code instrumentation language [32]. 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.

2.2 *Ad hoc* static analysis tools

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 only implements 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 [33], traversing the AST and control- and data-flow graphs.

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 [34]. They use existing analysis tools that derive metrics from their output, and add their own additional analyses and metrics. SonarQube finds not only bugs but also bad smells, which do not prevent correct program functioning, but usually correspond to another problem in the system [35] (e.g., code duplication, forgotten interfaces and orphan abstract classes). SonarQube can be extended with user-defined plug-ins, by implementing so-called *sensors* and *decorators*.

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) [36]. Coverity does not allow users to write their own analyses.

PMD is an extensible cross-language static code analyzer [37]. It finds common programming flaws, such as unused variables, empty `catch` blocks, unnecessary object creation and copy-pasted code. 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.

2.3 Classification of programmers by their expertise level

In the field of programmer classification, Lee *et al.* used biometric sensors data to detect the programmer's level of expertise [38]. In particular, they used a 16-channel-amplifier V-amp to collect electroencephalographic data, and a SMI RED-mx eye-tracker to classify eye movement according to the velocity of shifts in the programmer eye direction. They conducted a study with 38 expert and novice programmers, investigating how well electroencephalography and an eye-tracker data can be utilized to classify novice and expert programmers in two kinds of programming tasks (easy and difficult). By using Support Vector Machines, they built three models: one using the eye-tracker, another one with the electroencephalographic sensors, and the third one with both data sources.

The F1 performances of the three models to classify expert programmers were, respectively, 90.3%, 93.1% and 97%.

Samy S. Abu-Naser built an artificial neural network to classify the academic performance of students in linear programming [20]. Naser used the Linear Programming Intelligent Tutoring System (LP-ITS), created in the Al-Azhar University to teach linear programming. LP-ITS automatically generates programming problems for the students to solve [39]. The system stores information about the student interaction with LP-ITS, that is used to train the model. They used the log files of 67 learners, with 2144 program submissions; half for training and half for testing. A multilayer perceptron neural network (9 nodes in the input layer, 5 for the hidden one, and one node in the output layer) with sigmoid activation function was used to classify student's level of expertise. The average accuracy obtained was 92%.

2.4 Clone detection

Machine learning has been used to detect syntax patterns in source code for different purposes. One of the most active fields is clone detection, which is aimed at finding similarities between source-code fragments. Clone detection is used to identify repeated (not reused) code for software maintenance, program understanding, code refactoring, plagiarism detection and program compaction [40]. There exist different approaches to detect clones: text-based techniques, lexical, syntactic and semantic approaches, and hybrid methods [41].

The syntactic approaches to detect clones use parsers to convert source code into parse trees or ASTs, which are then processed using tree-matching or structural metrics [41]. Tree-matching techniques compare tree structures; whereas metrics-based systems gather metrics for code fragments and compare the metrics vectors, rather than the AST, to perform the classification [42].

CloneDr is a tree-matching clone detector that compares trees with the same structure [43]. To that aim, they propose three algorithms: one to detect subtree clones; another one for variable-size sequences of sub-trees; and the third algorithm to generalize combinations of other clones [44]. A combination of those three algorithms is used to compare tree structures. The tool was applied to a production software of more than 400K lines of code written in C, detecting average levels of source code duplication of 28%.

Wahler *et al.* find exact and parameterized code fragments, defining a method based on the concept of frequent itemsets, which works with an XML representation of ASTs [45]. In data mining, frequent itemsets are used to illustrate relationships within large amounts of data. For ASTs, frequent itemsets represent sequences of consecutive statements constituting source code clones. The implementation uses an XML database plus a link structure and a hash table to speed up the data access. It took 60 minutes to detect clones in the JDK source, and 90 minutes for the DisLog Developer's Kit (DDK) of SWI-Prolog [45].

Evans *et al.* built Asta, a clone detection system that works with so-called

structural abstractions (i.e., AST subtrees) [40]. Programs are parsed, and their ASTs are stored as XML documents. Subtrees are represented as patterns that are matched with other subtrees to detect potential clones. Those patterns are sometimes defined with holes, which are wildcards that allow any subtree match. This feature allows Asta to detect clones with variations in arbitrary subtrees. In a variety of 425K lines of code of Java and 16K lines of C#, 20-50% of the clones found by Asta were structural, and thus beyond lexical methods [40]. Its C++ implementation took 10 minutes to analyze all the code.

Koschke *et al.* defines a clone detection system based on abstract trees, but with linear time and space, similar to lexical alternatives [46]. The proposed algorithm parses the input program, creates the AST, serializes the trees, applies suffix tree detection, and decomposes the resulting token sequences into syntactic structures. Suffix tree comparison is linear in space and time with respect to their length. The proposed system, `cscope`, was tested with the Bellon benchmark, providing accuracies similar to syntactic approaches, with runtime performance of lexical ones [46].

The tool DECKARD computes certain characteristic vectors to approximate the structure of ASTs in a Euclidean space [47]. DECKARD identifies relevant and irrelevant nodes in the AST for clone detection. Characteristic vectors are created by counting the number occurrences of relevant nodes in a subtree. Then, DECKARD detects similar clones by computing locality sensitive hashing, which clusters similar vectors using the Euclidean distance metric. DECKARD was used to find clones in the open JDK and Linux kernel, performing better than CloneDr in accuracy and scalability [47].

2.5 Other scenarios of syntactic classifiers

Ahmad Taherkhani used decision tree classifiers to recognize sorting algorithms from Java code [48]. Algorithms are first converted into vectors of characteristics, including syntactic features such as the number of expressions, tokens, loops, algorithm length, and roles of variables. Then, a C4.5 decision tree classifier is built. The model is trained with five different types of sorting algorithms: Quicksort, Mergesort, Insertion sort, Selection sort and Bubble sort. Taherkhani collected 209 programs for the five sorting algorithms. The programs were gathered from various textbooks on data structures and algorithms, together with course materials available on the Web. A leave-one-out cross-validation technique was used to test the model. The average classification accuracy of the C4.5 model was 98.1%.

NATE is a tool to localize novice type errors in OCaml programs [49]. They convert the ASTs of source programs into Bags-of-Abstracted-Terms (BOATs), where each subtree is abstracted as a feature vector, comprising the numeric values returned by feature functions applied to the tree. NATE is trained with a large corpus of ill-typed programs and their "fixed" version, creating different models with distinct supervised-learning techniques. The resulting models take an ill-typed program and produces a list of potential blame assignments ranked by

likelihood. One important feature of NATE is that it just works with expression nodes in the AST. Thus, it does not support error prediction of any syntax construct different to expressions (e.g., statements, functions or data structures). NATE is able to predict correctly the exact sub-expression that must be changed 72% of the time.

There are some other scenarios where machine learning has been used to predict properties of programs by using syntax (and sometimes semantic) information of their source code. Some examples are vulnerabilities detection in source code [15], source-code decompilation [16, 50], code completion [51], and code idiom mining [52].

2.6 Structured methods

Alternative structured methods such as Graph Neural Networks operate on graph structures to provide node and graph classification [53]. In the context of program classification, Lu *et al.* conducted an experiment comparing Tree-Based Convolutional Neural Networks (TBCNN), Gated Graph Neural Networks (GGNN) and Gated Graph Attention Neural Networks (GGANN) [54]. They took C++ code from students, aimed at solving different programming tasks. They evaluated the performance of the three abovementioned neural networks to classify source code into two predefined categories (five similar programming tasks in each group). They created a graph representation that integrated ASTs with the semantic information of Function Call Graphs (FCG) and Data Flow Graphs (DFG). When only syntactic information (i.e., ASTs) was used in the classification, the average accuracies of TBCNN, GGNN and GGANN were, respectively, 94.1%, 96.4% and 97.1%. When semantic information was added, the performance improvement of the classifiers was not significant [54].

Syntax trees were also used with structured prediction methods such as Conditional Random Fields [12]. CRF is a probabilistic framework for labeling and segmenting structured data, such as sequences, trees and graphs [55]. CRFs define conditional probability distributions over label sequences, given particular observation sequences. CRFs was used to implement JSNice, a tool that predicts names of identifiers and type annotations of variables in obfuscated JavaScript code [12]. By analyzing the usage of variables in a function body, JSNice is able to label the type of the variable (if it is built-in) and a suitable name. In the evaluation of JSNice, it predicted correct names for 63% of the identifiers, and its type annotation predictions were correct in 81% of the cases [12]. A limitation of CRF is that it suffers high computation and space costs to be used with massive codebases [56].

Relational learning is another technique already identified as a mechanism to undertake graph mining [57]. Inductive Logic Programming (ILP) is a relational machine learning technique that can be used to classify graphs. Applied to programming language analysis, Sivaraman *et al.* implemented ALICE, an ILP tool to search source code by specifying syntax constructs [58]. The syntax of the language to be queried is specified as logic facts, representing tree structures.

Users define their queries by annotating pieces of code with similar structure to the one to be searched. After executing the query, irrelevant examples could be labeled by the user. Then, ALICE learns the query that includes the program structure, by using ILP. When labeling two positive examples and three negative ones, ALICE successfully identifies similar code locations with 93% precision and 96% recall in 2.7 search iterations [58]. Some other performance experiments have indicated that ILP does not seem to scale well with big data [59].

Tree kernels measure similarity between two trees in terms of their sub-structures [60]. Besides natural language processing, tree kernels have been used for source-code plagiarism detection [61]. WASTK is a tool that builds the ASTs of two programs and gets the similarity between them by computing the tree kernels of both trees. In WASTK, Term Frequency-Inverse Document Frequency (TF-IDF) weights are assigned to each node to avoid the misjudgment caused by common code snippets. In the experiment conducted by Deqiang Fu *et al.*, the average performance of WASTK was 48.32%, significantly higher than the JPlag and Sim software plagiarism tools [61]. Training time of tree kernels is quadratic in the size of the trees [62].

Chapter 3

ProgQuery Infrastructure

ProgQuery is an efficient and scalable infrastructure that allows the user to perform operations such as program analyses, advanced queries and program feature extraction. Such operations are 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 the ProgQuery open-source infrastructure, users could write their own queries and analyses using an expressive declarative query language.

In this chapter we describe the architecture of ProgQuery (Section 3.1). Then, we detail the design of the Java compiler plug-in developed (Section 3.2), and a brief description of the different language representations defined (Section 3.3)—a formal definition is presented in Appendix A. Forthcoming Chapter 4 presents a web application built over the ProgQuery infrastructure, and the implementation of different analyses together with an evaluation of ProgQuery are depicted in Chapter 5.

3.1 Architecture

Figure 3.1 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 with other different semantic representations through semantic relationships, and vice versa (Section 3.3). 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 [63], check for guideline compliance [18], search for code using advanced queries [9], obtain software metrics [64], and extract datasets with syntactic and semantic information [17]. Our platform provides a collection of existing services (analyses, queries,

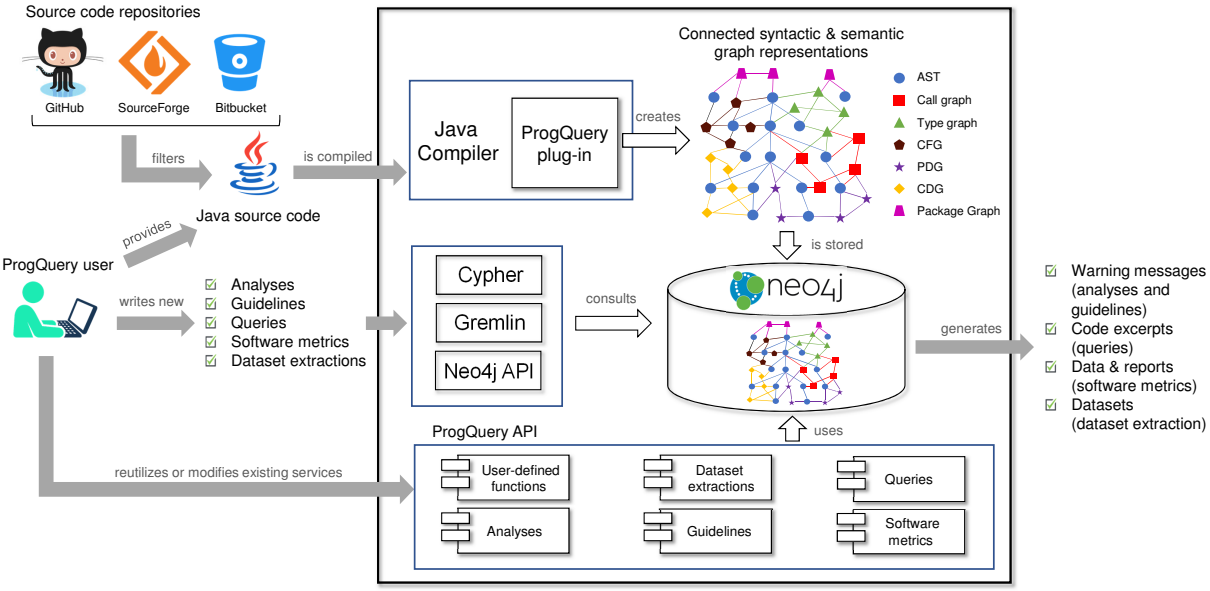


Figure 3.1: Architecture of ProgQuery.

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 1.2).

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) [65]. 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 [66].

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) returned. Software data and reports are the output for metrics requests, and datasets are returned for queries requesting syntactic and semantic information.

3.2 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 3.3.

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 [67].

The annotated AST is traversed with the *Visitor* design pattern [33]. 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.

3.3 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 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 1.2.

Figure 3.2 shows a small excerpt of the seven graph representations created by ProgQuery for the source code in Figure 1.1. What follows is a brief description of such representations (detailed information can be consulted in Appendix A).

3.3.1 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 3.2, 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`

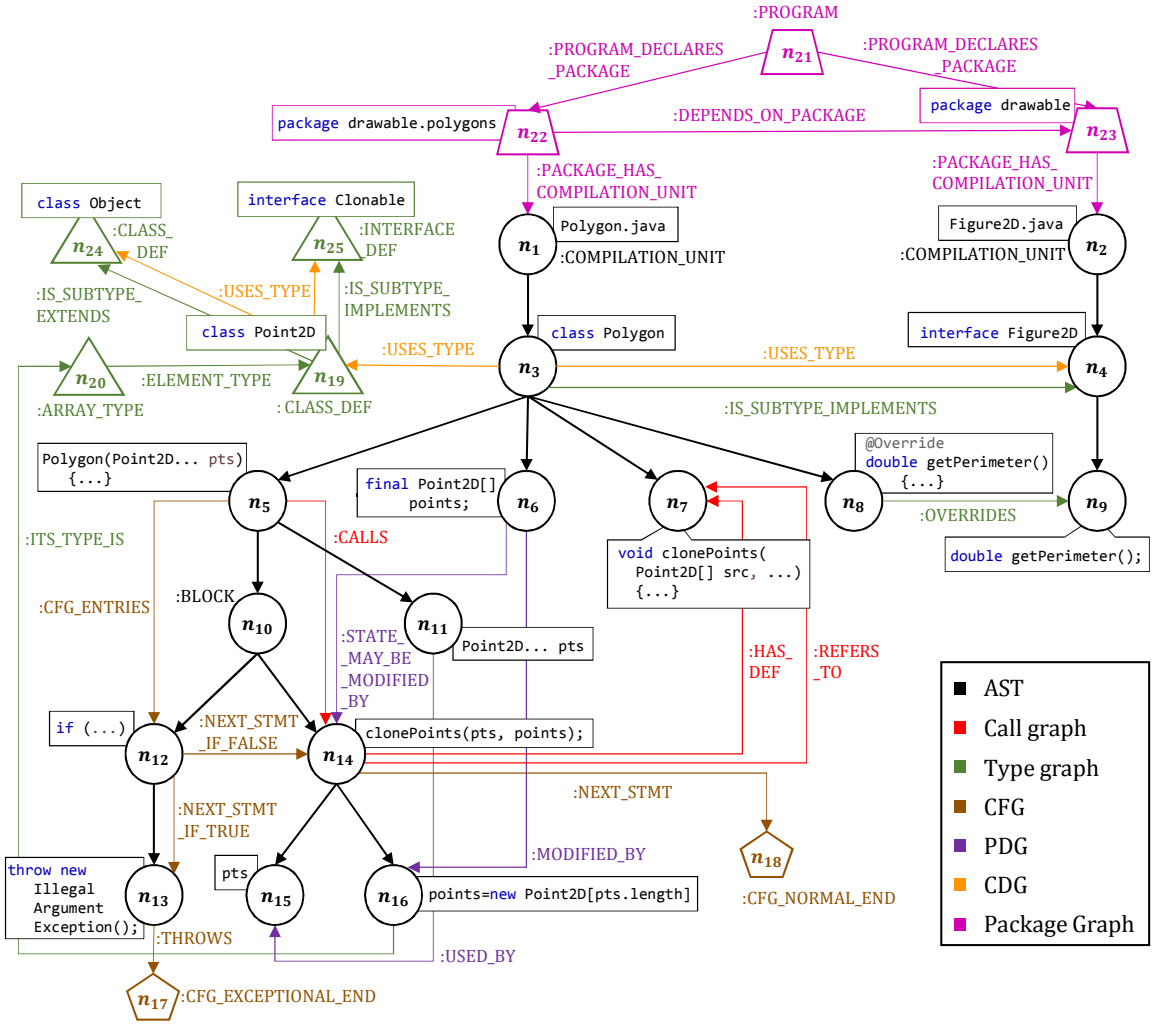


Figure 3.2: Seven different graph representations for the Java program in Figure 1.1.

and `BLOCK` are two example labels depicted in Figure 3.2[1]. We also use the multiple label capability provided by Neo4j to allow common generalizations of AST nodes [67]. 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 5.4.2).

Syntactic relationships between nodes are also labeled in Neo4j. For example, node n_1 in Figure 3.2 is connected to node n_3 through a `HAS_TYPE_DEF` labeled relationship[2].

3.3.2 Control Flow Graph (CFG)

CFGs represent the execution paths that may be traversed when the program is run. First, it connects method or constructor definitions to their first statement

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

[2]All the labels are not shown; details in Appendix A.

(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 1.1 (n_{14} in Figure 3.2) 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).

CFG also models the exceptional jumps performed by Java checked exceptions[1], and assert and throw instructions. For that purpose, ProgQuery defines CFG nodes representing exceptional method termination, and exceptions handled in a 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. In Figure 3.2, the throw statement represented by n_{13} is connected to the CFG_EXCEPTIONAL_END semantic node n_{17}, because no catch or finally blocks are used to handle the exception.

3.3.3 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) may be read or modified. In Figure 3.2, 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 be changed. This is a valuable information for many analyses that consider object mutability [63]. For 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 may 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 1.1).

3.3.4 Call Graph

This representation provides information about method and constructor invocations in the source code. Method/constructor definition nodes are connected

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

through CALLS relationships to all the invocations in their bodies. In Figure 3.2, 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).

3.3.5 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.

3.3.6 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.

3.3.7 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 3.2), 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 3.2 represents the root node for the source code in Figure 1.1. 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.

Chapter 4

Web Prototype

In this chapter, we describe a web prototype we developed to show how the ProgQuery infrastructure can be used to provide users with a mechanism to create and share their own static analyses for Java programs. Its purpose is to illustrate the potential of ProgQuery, and help us to understand the future research works presented in Chapter 8. We first describe the architecture of the system, detailing each element afterwards.

4.1 Architecture

Figure 4.1 shows the architecture of the web prototype. The system provides its services by means of two interfaces. The first one is a web application that could be accessed by users with any web browser. The second one is a Web API (a collection of REST web services), aimed at constructing client applications that consume the services provided by ProgQuery.

Programs to be analyzed may be uploaded by the user or a client application. They could also be taken from an existing source code repository, such as GitHub SourceForge and Bitbucket (*repository management* component in Figure 4.1). When a program is included in ProgQuery, it is compiled by the system. The infrastructure provides a Java compiler plug-in that modifies the compilation process (Section 3.2). ProgQuery also provides another Maven plug-in that creates the program representations described in Section 3.3, by modifying the way Java applications are compiled with Maven. Those representations are created by the infrastructure (used by both the Maven and Java compiler plug-ins), and stored in a Neo4j database.

New analyses could be added and shared with the existing users. Analyses are written in Cypher. They could be typed in the web application or uploaded to the server. An analysis could be executed against one or many programs, and the results are stored in the system. The output of all the previous analyses run by the user can be consulted anytime. Different user information is also stored in the system. The business logic related to these entities is placed in the *analysis, program and user management* module in Figure 4.1. A relational database is used to store those entities.

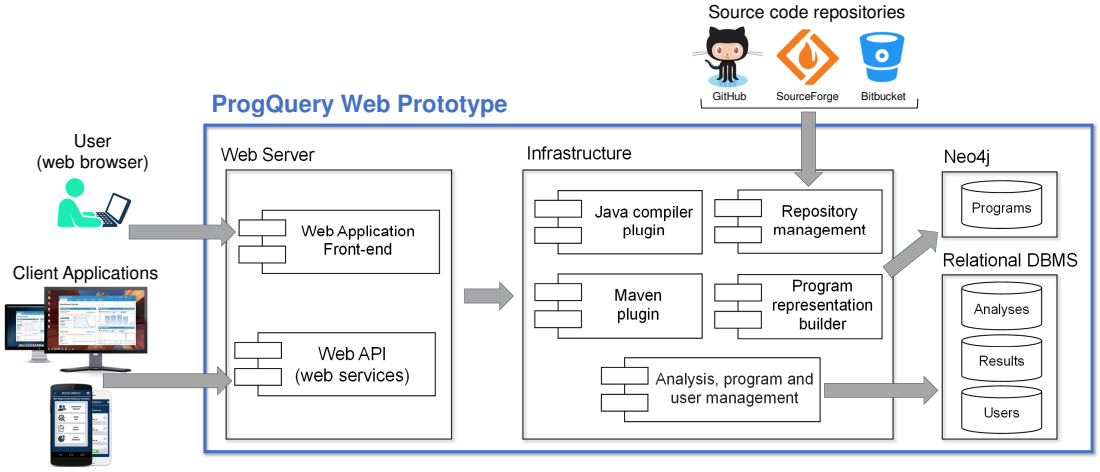


Figure 4.1: Architecture of the web prototype.

4.2 Web application

We briefly describe the functionalities of the web application, classifying them by the domain entities.

4.2.1 Programs

As shown in Figure 4.2, the prototype currently allows uploading a single Java file (to test basic analyses), a Java project with a zip file, or an existing project in GitHub. For the single file and zip options, the Java compiler plug-in is used, and any parameter could be passed to the Java compiler. Notice that the zip file could provide any library, by including their jar files in the compressed archive to be uploaded.

If the zip file contains a `pom.xml` file, the Maven plug-in can be used for the compilation. If the *maven compile* command is executed without errors, program representations are included in ProgQuery. With this method, Maven transparently downloads all the program dependencies (libraries) declared in `pom.xml`, so they do not need to be included in the zip file.

The third option allows to take a program from GitHub, compile it and upload it to ProgQuery. The GitHub project must provide a Maven `pom.xml` file fulfilling the requirements described in the previous paragraph. Alternatively, if the project contains all its dependencies it could be compiled with the Java compiler.

All the Java projects uploaded by a user can be listed, consulted and deleted by that user. When a project is uploaded (with any of the three options provided), the user may optionally select existing analyses to be run against the uploaded program.

4.2.2 Analyses

A naming system was defined to allow the utilization of numerous analyses in ProgQuery. For this purpose, we selected the Reverse Domain Notation (RDN)

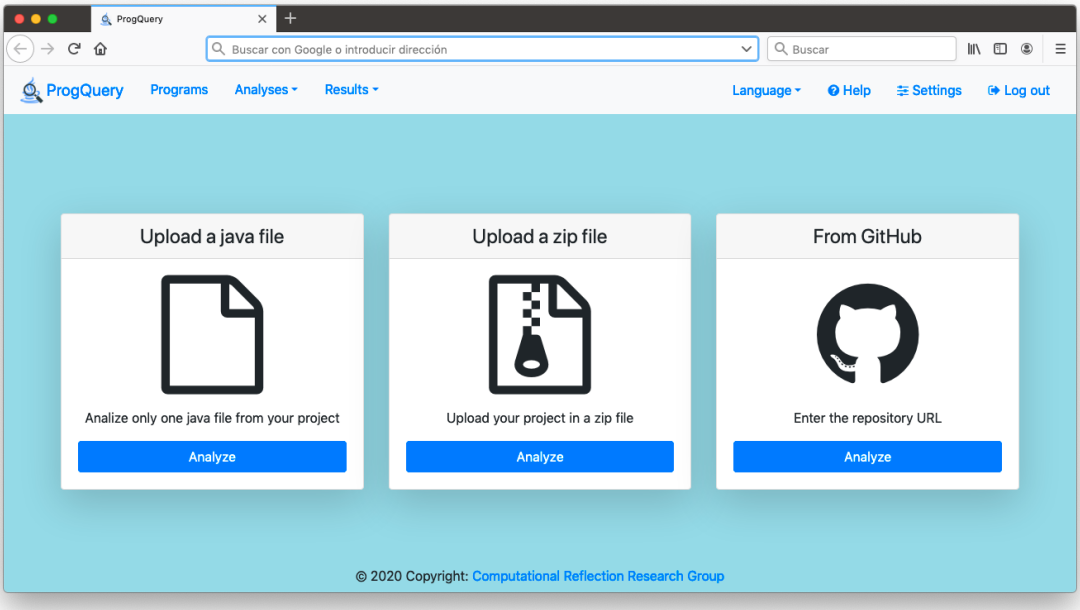


Figure 4.2: Java program upload.

naming convention, based on reversing registered domain names. For example, the motivating analysis presented in Section 1.2.1 is registered as `es.uniovi.reflection.cmu.OBJ-50`, since it implements the OBJ-50 recommendation by CERT division at CMU [18], coded in ProgQuery by the Computational Reflection research group of the Spanish University of Oviedo. We propose the same naming convention for programs (Section 4.2.1).

Our system provides the $*$ wildcard to facilitate analysis search and grouping. For example, `es.uniovi.reflection.cmu.*` represents all the CMU analyses implemented by the Reflection research group at University of Oviedo. Likewise, `es.uniovi.reflection.*` selects all the analyses developed by that research group.

New queries could be added by writing them in a textbox or uploading existing files. At insertion, its name, description and visibility should be provided. Visibility can be private (just for the current user), public (for everyone) or shared with existing users. When a new analysis is created, the user can also specify a set of programs to execute the analysis against.

The 13 analyses described in Section 5.1 are provided to the user (we plan to add more analyses). We think this is a valuable service, because novice users may not be fluent in the use of ProgQuery. In that case, analyzing existing queries will be helpful. They can also modify those analyses to adapt them to new requirements.

Figure 4.3 shows one way to run analyses. The user may search for an existing project in ProgQuery. For basic tryouts, the Java code to be analyzed may be written in the big textbox component on the left-hand side. The same options are available for analyses: we can search an existing one or type its Cypher code.

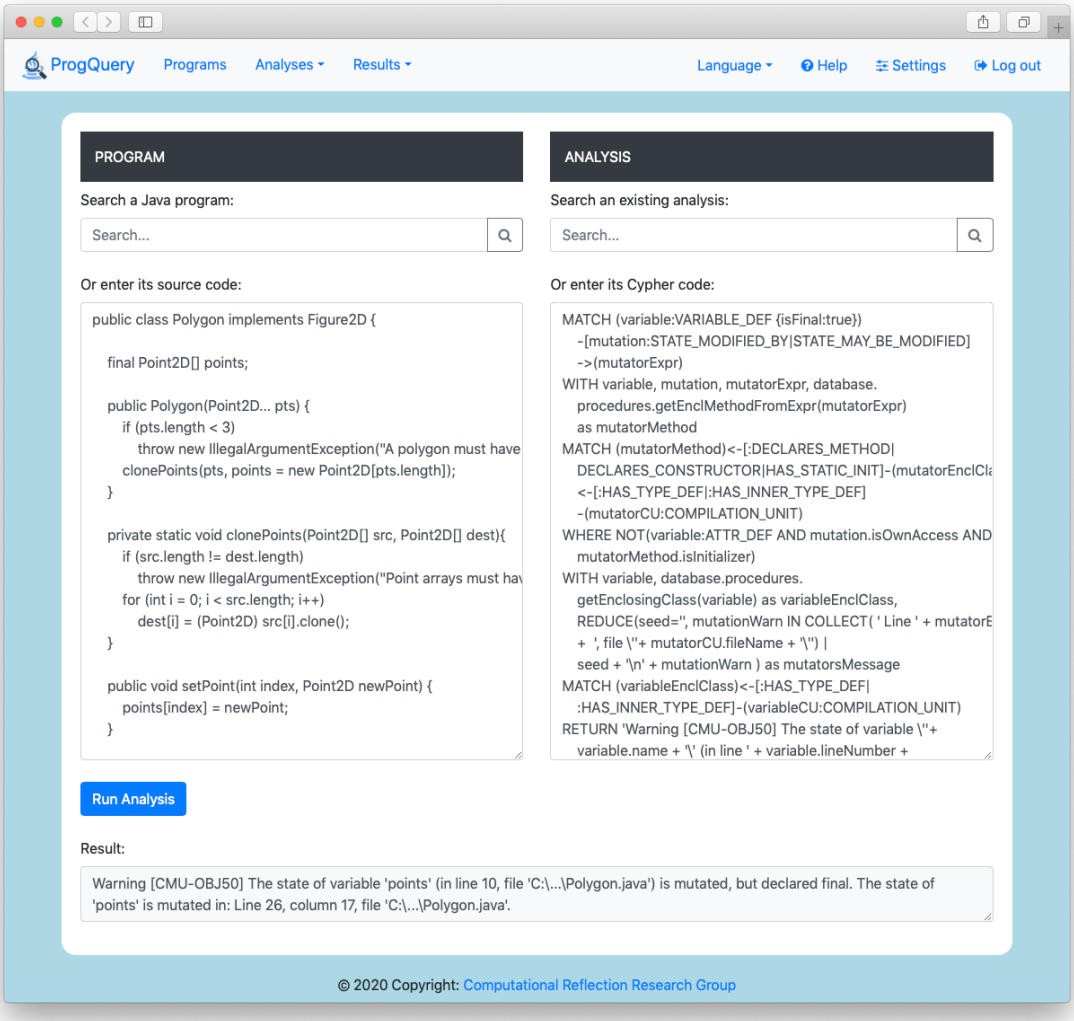


Figure 4.3: Analysis execution.

The results window is used to display the output of the analysis or the error messages of either the compilation process or the analysis execution. Another view of the web application allows the execution of multiple analyses against multiple programs.

Analyses stored in ProgQuery could be searched (using wildcards), modified and deleted (if the user is the owner).

As defined in Appendix A, ProgQuery also provides a set of user-defined functions to make it easier the development of new analyses. For example, the `database.procedures.getEnclosingStmt` function returns the enclosing statement where a particular expression is used; and `database.procedures.getEnclosing-Method` returns the method where a statement has been written.

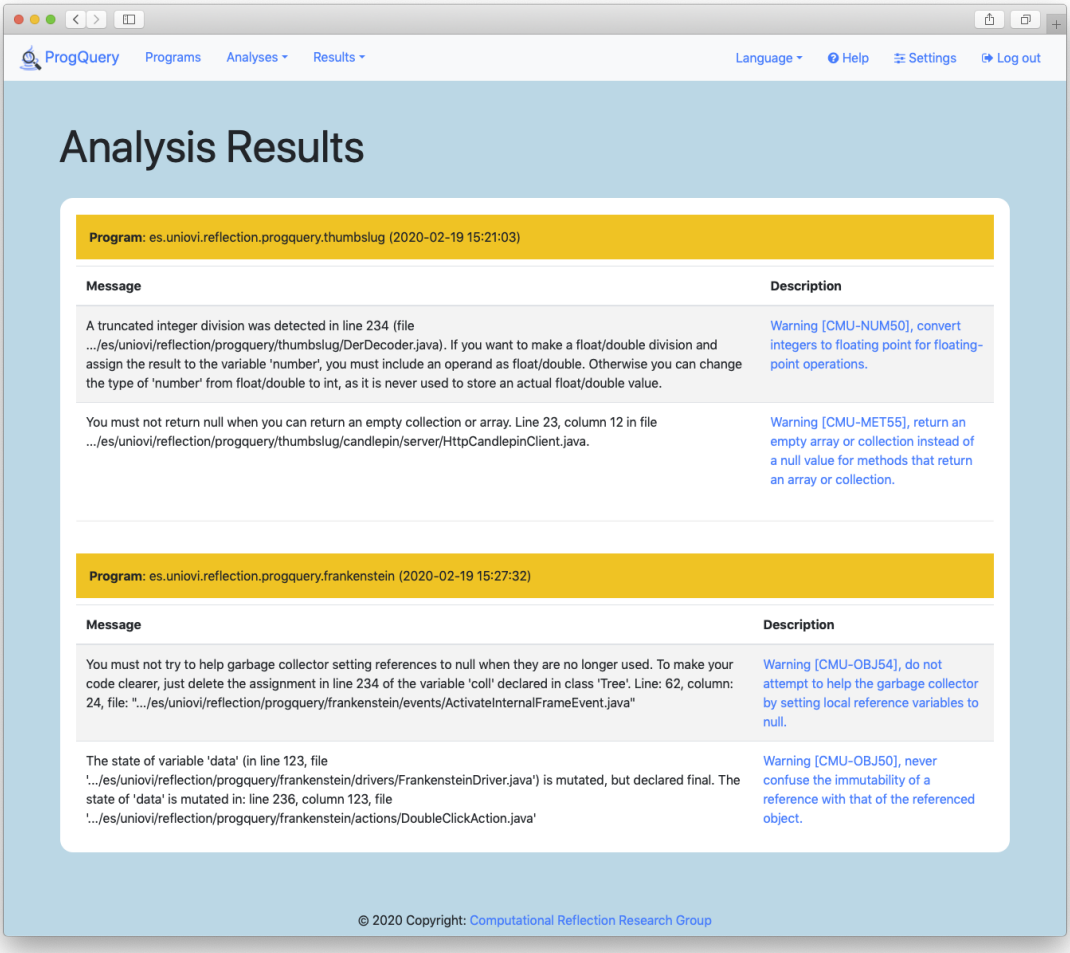


Figure 4.4: Results of executing an analysis against a given program.

4.2.3 Results

As mentioned, ProgQuery stores the results of every analysis executed by the logged user. If the execution is successful, the user will be able to consult all the messages produced by the analyses, their date and time of execution, the analyses run and programs analyzed.

Figure 4.4 shows one example view of results. Different analyses were executed against different programs. The first program (Thumbslug) has warnings, produced as the result of executing some of the analyses described by the CERT division of Carnegie Mellon University [18]. Afterwards, ProgQuery shows the warnings generated by the analysis of the Frankenstein program.

The *results* menu also provides information about the analyses under execution, which are yet to be completed. It is important to notice that some analyses may take long computation time, especially when they are executed against big Java projects. The user may force termination of analyses being executed, if he/she is the owner. Likewise, the results of finished analyses can also be deleted.

4.2.4 Users

New users could be created by specifying their email, password and given and family names. This information can be changed by users, once they are logged in the system (settings menu). Users can unsubscribe themselves from the system.

4.3 Web API

As mentioned, ProgQuery also provides its functionality as a collection of web services. By using the Web API provided, client applications could be developed.

All the services provided require user authentication. We use JSON Web Tokens (JWT) to authorize users and protect API routes [68]. JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties [69]. The JWT-based authentication mechanism allows users to enter their username and password in order to obtain a token, which allows them to access Web API resources for a time period.

When the user accesses any service, the Web API server redirects him/her to the authentication service, if there is no token or the token is expired. Users send their credentials (login and password) to the authentication service. This service validates the credentials and generates a JWT containing the user details and the expiration time. Finally, it signs the token content with a secret key before sending it to the client as a response.

Whenever the user wants to access any service, the user agent must send the JWT in the authorization header [70]. Then, the Web API server takes the JWT from the message header, decodes it, validates the signature, verifies user permissions, executes the service, and returns the service response. If such authentication fails, the Web API sends an HTTP code 401, unauthorized response.

Moreover, we only provide our services over SSL+HTTPS to encrypt the communications. In this way, potential attackers that could sniff the network traffic would not obtain the JWT used by the user.

We now provide a brief description of the web services provided by ProgQuery.

4.3.1 Programs API

- `GET /api/programs`. Only available for administrators. Service that lists all the programs stored in the system.
- `GET /api/programs/{id}`. Returns the program with `{id}` Reverse Domain Name (RDN).
- `GET /api/programs?user={user}`. Lists all the programs inserted by `{user}`.
- `POST /api/programs`. Inserts a new program in the database, and returns it. The arguments to be passed are:
  - RDN of the program.

    - The name of the user inserting the program.
    - Compilation mode/strategy (i.e., Java compiler or Maven).
- `PUT /api/programs/{id}`. Updates the source code of one given program, passing its `{id}` RDN and the compilation mode/strategy.
- `DELETE /api/programs/{id}`. Deletes the program with `{id}` RDN and returns it.

4.3.2 Analyses API

- `GET /api/analyses`. Lists all the analyses. Only available for admin users.
- `GET /api/analyses/{id}`. Gets the analysis with `{id}` RDN.
- `GET /api/analyses?user={user}`. Lists the analyses the given `{user}` has access to.
- `GET /api/analyses?user={user}&owner=true`. Lists the analyses whose author is `{user}`.
- `POST /api/analyses`. Creates a new analysis and returns it. The arguments to be passed are:
    - RDN of the analysis.
    - User who owns the analysis.
    - A description of the analysis.
    - The Cypher code implementing the analysis.
    - Visibility (private, public or shared among a list of users).
- `PUT /api/analyses/{id}`. Updates the analysis with `{id}` as RDN. The analysis attributes that could be updated are analysis name (RDN), description, Cypher code and visibility (private, public or shared).
- `DELETE /api/analyses/{id}`. Deletes the analysis with `{id}` RDN and returns it.

4.3.3 Results API

- `GET /api/results`. Only available for administrators. This service lists all the results stored in the system.
- `GET /api/results/{id}`. Gets the result with `{id}` as identifier.
- `GET /api/results?programId={programId}`. Lists the results stored for the `{programId}` RDN.
- `GET /api/results?analysisId={analysisId}`. Lists the results stored for the `{analysisId}` RDN.
- `GET /api/results?user={user}`. Lists the stored results for the analyses executed by `{user}`.

- `GET /api/results?programId={programId}&analysisId={analysisId}`. Lists the stored results relative to {`analysisId`} (RDN) and executed against {`programId`} (RDN).
- `POST /api/results`. Executes an analysis against an inserted program and returns the obtained result. The arguments to be passed are:
  - RDN of the analysis.
  - RDN of the program.
- `PUT` is not allowed for this entity.
- `DELETE /api/results/{id}`. Deletes the result {`id`} and returns it.

4.3.4 Users API

- `GET /api/users`. Service that lists all the users registered in the system. Only available for admin users.
- `GET /api/users/{user}`. Gets the user with {`user`} identifier (emails are used as user identifiers).
- `GET /api/users/{user}/programs`. Lists the programs inserted by {`user`}. This service identifies the same resource as `GET /api/programs?user={user}`.
- `GET /api/users/{user}/analyses`. Lists the analyses whose author is {`user`}. This service identifies the same resource as `GET /api/analyses?user={user}&owner=true`.
- `GET /api/users/{user}/results`. Lists the stored results of the analyses executed by {`user`}. This service identifies the same resource as `GET /api/results?user={user}`.
- `POST /api/users`. Registers a new user in the system and returns it. The arguments needed to register a new user are: email address, name, surname and password.
- `PUT /api/users/{user}`. Updates the user with {`user`} identifier (email). The user attributes that could be updated are: email address, name, surname and password.
- `DELETE /api/users/{user}`. Deletes the user with {`user`} as email and returns it.

Chapter 5

Use Case Scenario 1: Static Program Analysis

The objective of this chapter is twofold. Firstly, it shows how ProgQuery, with the ontology defined in Appendix A, can be used as an infrastructure to define advanced program analysis declaratively (Section 5.1). Secondly, the collection of analyses are used to evaluate the runtime performance, scalability and memory consumption of ProgQuery, and compare it with related systems. Different research questions are raised to perform that evaluation (Section 5.2).

5.1 Static analyses

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 [71]. Among other tasks, CERT identifies coding practices that can be used to improve the quality of software systems, classifying them as rules or recommendations [72]. These coding practices are taken from different programming language experts and other sources such as [63], [73] and [74]. 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 (its source code can be consulted in Appendix B, and downloaded from [75]).

1. MET53-J (program understandability) [76]. *Ensure that the* `clone` *method calls* `super.clone()`. `clone` may call another method that transitively calls `super.clone()`.

2. MET55-J (reliability) [63]. *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) [73]. *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) [63, 73, 76]. *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) [63, 76]. *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) [74, 77]. *Avoid cyclic dependencies between packages.* Cyclic dependencies cause issues related with testing, maintainability, reusability and deployment.

7. OBJ54-J (programming misconceptions) [63]. *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) [63, 73]. *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 1.2.1).

9. ERR54-J (reliability) [73]. *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) [78]. *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) [63]. *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 programmer to use local variables instead.

12. OBJ56-J (security) [79]. *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) [73]. *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.

5.2 Research questions

As mentioned, the analyses described in the previous section will be used to evaluate ProgQuery and related systems. To that aim, we conduct various experiments to address the following research questions:

1. Does the proposed system provide lower analysis times than related approaches?
2. Does it provide better scalability for increasing program sizes?
3. Does it provide better scalability for increasing analysis complexity?
4. Is it able to perform complex analyses against huge Java programs in a reasonable time?
5. Can program analyses be expressed succinctly, in a declarative manner, and using standard query-language syntax?
6. Are there any drawbacks of our system, compared to related approaches?

5.3 Methodology

We present the selected systems and programs 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.

5.3.1 Systems measured

We included in our evaluation different systems related to our approach. The analyses described in Section 5.1 were coded for such systems in order to compare them (the source code can be downloaded from [75]). 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 using 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 via RESTful services [80].

– Semmle CodeQL 1.20, a code analysis platform to perform detailed analyses of source code [22]. Semmle allows writing queries in QL, an object-oriented variant of the Datalog logical query language [81]. 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 [82]. 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.

5.3.2 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 [83]. 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 [84], API mining [85], and database framework usage analysis [86].

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 5.1 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 (NFEGuardian-Shared) is at the 92^{th} percentile.

5.3.3 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.* [87]. 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)

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

Table 5.1: Programs selected from the CUP GitHub Java corpus (percentile and position refer to the non-empty lines of code).

is approximately Gaussian distributed, provided that the samples x_i are independent and they come from the same population [87]. Therefore, taking $p = 30$, we can compute the confidence interval $[c_1, c_2]$ using the *Student's* t-distribution as [88]:

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

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 [87].

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) [89]. 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 [90].

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.

5.3.4 Experimental environment

To measure the systems described in Section 5.3.1, 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 5.1 shows an example of how two variables influence on runtime performance of the system. Above, it is showed how the heap size of our modification of the Java compiler influences on insertion time (Neo4j embedded). Below, Figure 5.1 illustrates how insertion times in Neo4j server depend on the number of insertions per transaction done by ProgQuery and Wiggle. Orange lines in Figure 5.1 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:

- 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).
- 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.
- Maximum (`-Xmx`) and initial (`-Xms`) heap memory used by the Java Virtual Machine when executing queries: we set both variables to 1 GB.

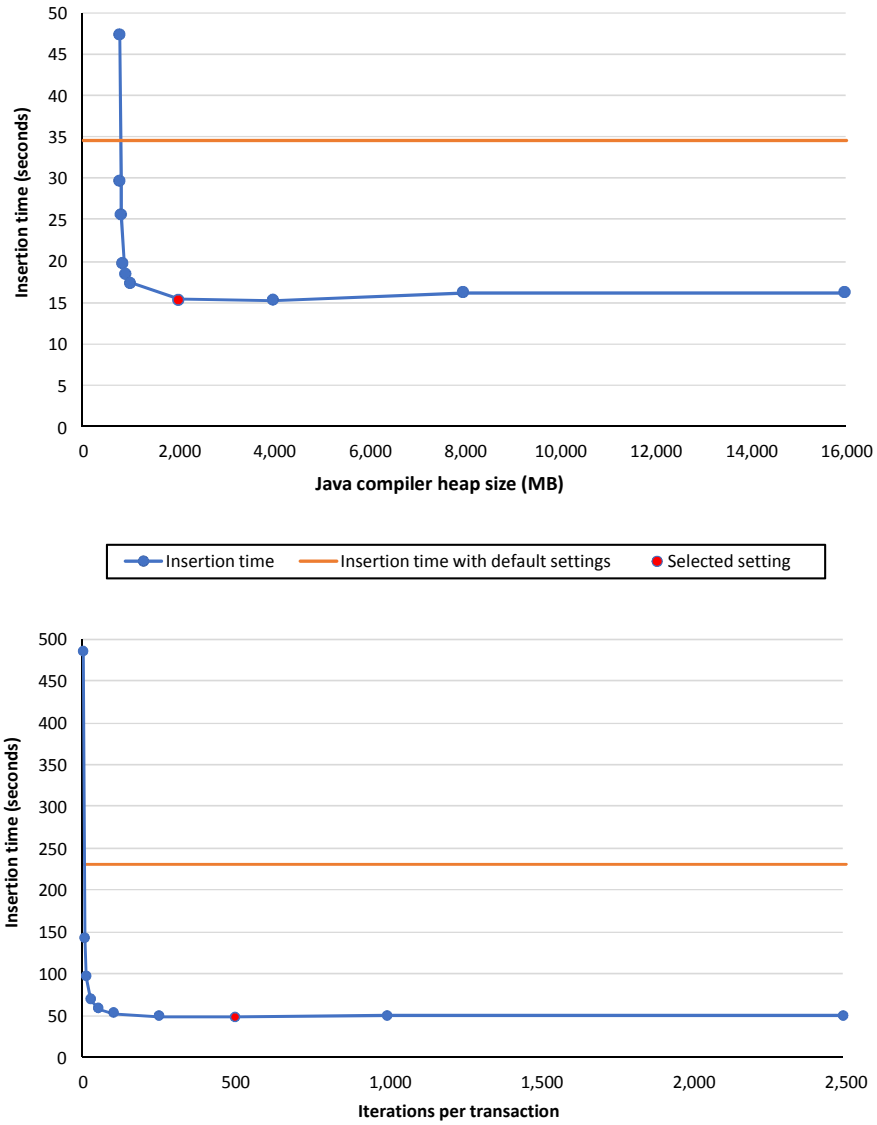


Figure 5.1: Dependency of Java compiler memory (above) and iterations per transaction (below) on insertion time for Neo4j embedded (above) and server (below), when inserting the Frankenstein program in Table 5.1.

– For insert operations, the startup memory used by our modified Java compiler (`-J-Xmx` and `-J-Xms`): 2 GB.

– Memory for running queries of Semmle CodeQL: 1 GB.

5.4 Evaluation

5.4.1 Analysis time

We measure and compare runtime performance of the systems described in Section 5.3.1. Their scalability related to program size and analysis complexity is also discussed.

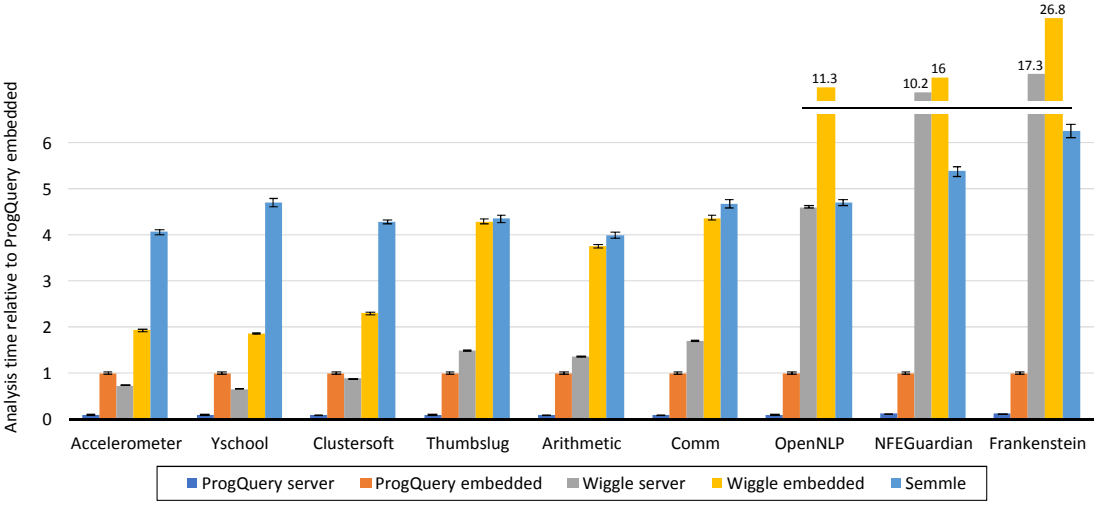


Figure 5.2: Average analysis execution time for increasing program sizes (execution times are relative to ProgQuery embedded).

5.4.1.1 Increasing program sizes

The first comparison, illustrated by Figure 5.2, 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 5.1, 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 5.3 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 5.3 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.

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.

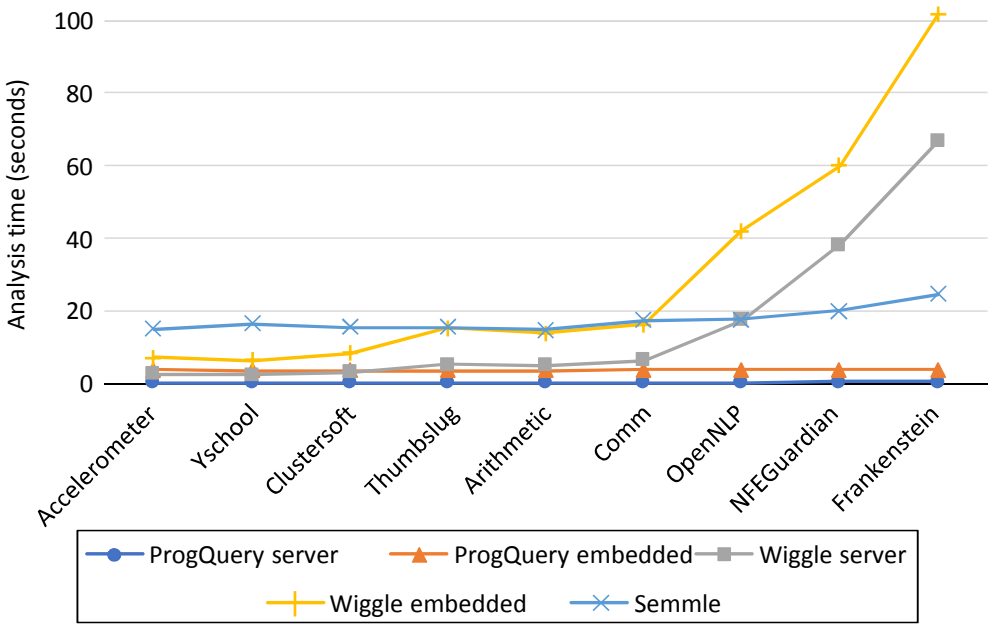


Figure 5.3: Execution time (seconds) trend for increasing program sizes (values shown are the geometric mean of execution times for all the analyses executed against the given program).

These data provide a response to **Research Question 2** (Section 5.2): ProgQuery scales significantly better than the other systems to increasing program sizes (we answer Research Question 1 in the following subsection).

5.4.1.2 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 3.3) consulted in the analysis. Therefore, we identify three levels of complexity:

- Easy. The analysis uses the syntactic representation (AST) and at most another semantic representation.
- Medium. The AST plus two other semantic representations are queried in the analysis.
- Complex. The analysis uses the AST and three or more semantic representations.

Table 5.2 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 5.4 shows the execution times for the three levels of complexity identified. Values shown are the average analysis times for all the programs in Section 5.3.2. 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

Analysis	AST	Call Graph	Type Graph	CFG	PDG	CDG	Package Graph	Complexity
DCL56-J	×							Easy
MET50-J	×							Easy
MET52-J	×				×			Easy
MET55-J	×		×					Easy
NUM50-J	×				×			Easy
SEC56-J	×		×					Easy
MET53-J	×	×	×					Medium
OBJ54-J	×			×	×			Medium
OBJ56-J	×		×		×			Medium
DCL53-J	×		×	×	×			Complex
DCL60-J	×		×		×	×	×	Complex
ERR54-J	×		×	×	×			Complex
OBJ50-J	×	×	×		×			Complex

Table 5.2: Program representations used by the different analyses and their level of complexity.

performs better (54%) than ProgQuery embedded, only for easy analyses. For the remaining cases, ProgQuery embedded outperforms the other systems.

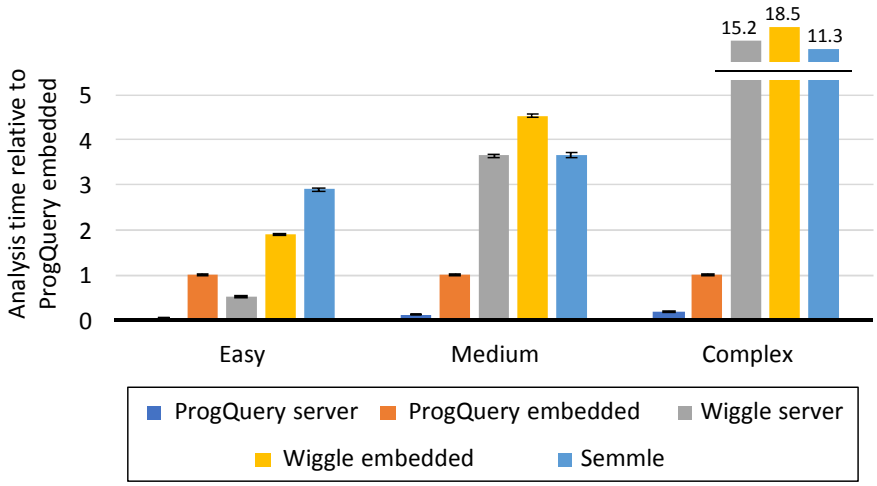


Figure 5.4: Average execution time for increasing analysis complexity (execution times are relative to ProgQuery embedded).

Therefore, the answer to **Research Question 1** (Section 5.2) 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 also use the same kind of persistence store (i.e., Wiggle embedded and Semmle).

Figure 5.5 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 3** (Section 5.2) is that ProgQuery scales significantly

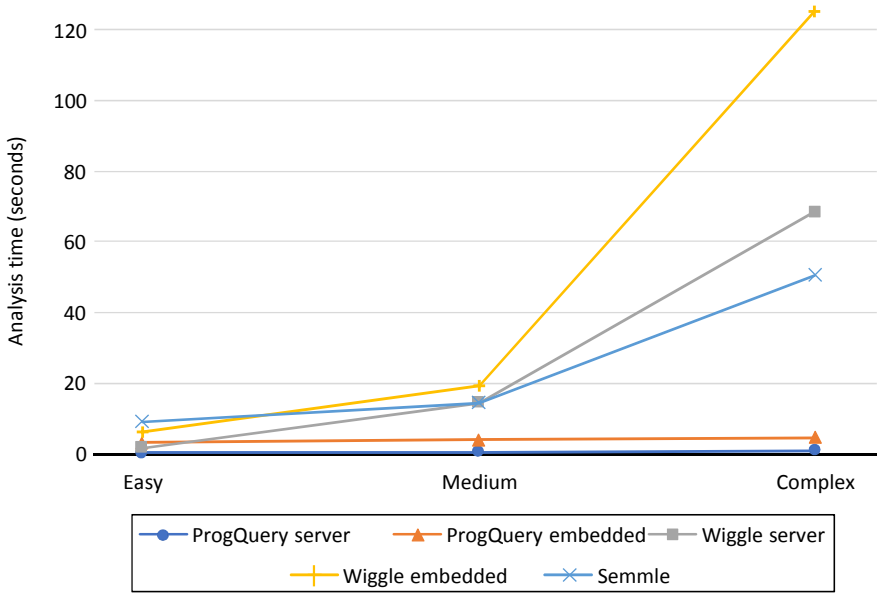


Figure 5.5: Execution time (seconds) trend for increasing analysis complexity (values shown are the geometric mean of execution times of all the programs of the given complexity).

better than Semmle and Wiggle for analysis complexity.

For easy 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 3.3). Finally, it seems that Semmle manages to reduce the number of accesses to the relational database in complex analyses, performing better than Wiggle.

5.4.1.3 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 5.1 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 5.3 shows ProgQuery analysis times for all the queries, under the configuration settings described in Section 5.3.4. That table gives us the response to **Research Question 4** (Section 5.2): 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.

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

Table 5.3: ProgQuery execution time (seconds) for analyses in Section 5.1, run against a Java program of 18M lines of code.

We can see in Table 5.3 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. 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.

5.4.2 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 [91].

Semmle provides the QL object-oriented declarative query language [81]. Its syntax is similar to SQL, but its semantics is based on Datalog, a declarative logic programming language [23]. 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 [25]. 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 5.4 presents the number of tokens (lexical elements), AST nodes and lines of code of the queries used to implement all the analyses for the three

	Analysis	Tokens			AST nodes			Lines of Code		
		PQ	Semmle	Wiggle	PQ	Semmle	Wiggle	PQ	Semmle	Wiggle
Easy	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 (easy)	**144**	**167**	**471**	**65**	**87**	**202**	**5**	**12**	**17**
Medium	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 (medium)	**239**	**374**	**1,656**	**117**	**195**	**814**	**9**	**33**	**45**
Complex	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 (complex)	**241**	**823**	**2,334**	**115**	**440**	**1,149**	**8**	**84**	**75**
	Mean (total)	**190**	**329**	**1,030**	**88**	**173**	**476**	**7**	**27**	**34**

Table 5.4: Number of tokens (lexical elements), AST nodes, and lines of code of the queries used to write all the analyses in the different systems (PQ stands for ProgQuery).

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 3.3), 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 to complex complexities, unlike the other systems (Table 5.4).

After this study about expressiveness, we can answer **Research Question 5** (Section 5.2). 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.

5.4.3 Memory consumption

Figure 5.6 shows the average RAM memory consumed by the five systems evaluated, under the configuration settings described in Section 5.3.4. 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 mem-

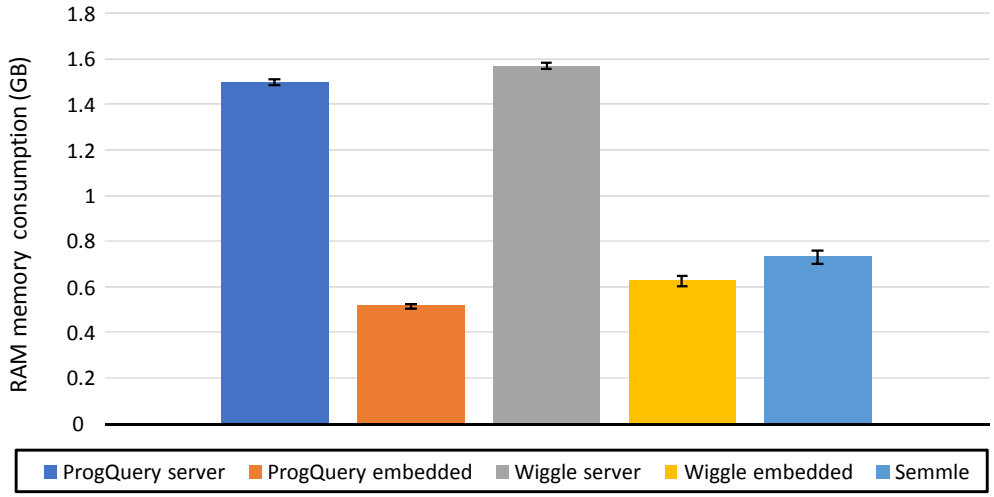


Figure 5.6: RAM memory consumed at analysis execution.

ory 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 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.

5.4.4 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 [92]. The user provides a code repository identifier, and LGTM compiles the program, returning the relational database file containing the program representation.

In Figure 5.7, 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 and Wiggle server decreases. This is because of an optimization we develop at compile time, based on avoiding the use of reflection [93] performed by Wiggle (Section 2.1).

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.

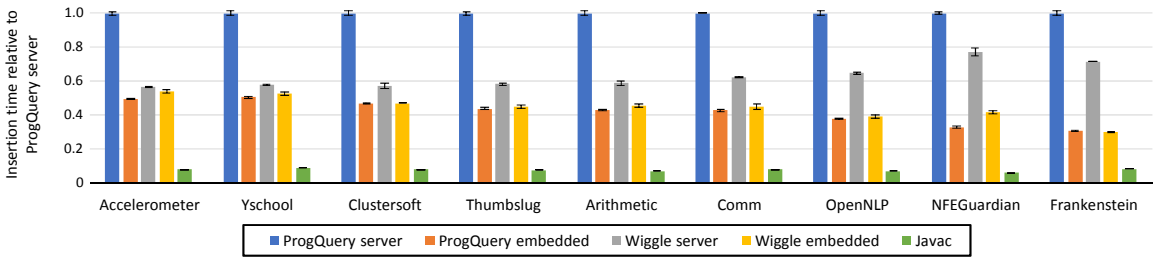


Figure 5.7: Insertion times for increasing program sizes, relative to ProgQuery server.

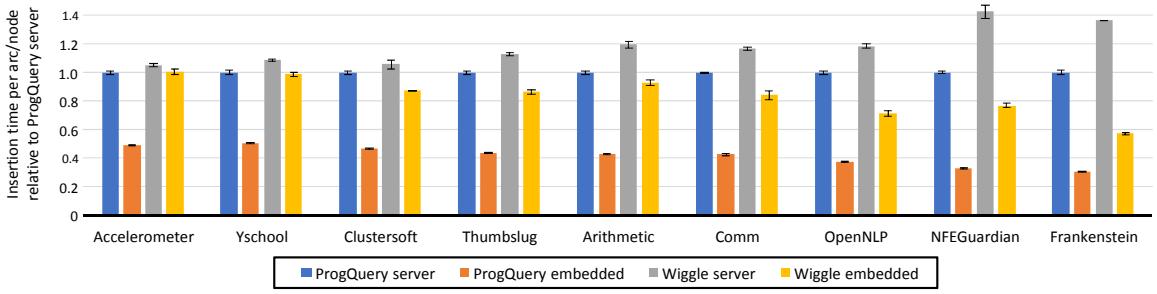


Figure 5.8: Insertion times per node/arc for increasing program sizes, relative to ProgQuery server.

Figure 5.7 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 server, and ProgQuery embedded and server.

Figure 5.8 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 6 (Section 5.2) 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%, and but this drops as program size grows. Another minor drawback is database size, which shows increases from 25% up to 97%.

Chapter 6

Use Case Scenario 2: Programmer Classification

In this chapter, we describe how we used the ProgQuery compiler plug-in to extract syntactic information, used to classify and score the level of programming expertise of developers, by analyzing the source code they write [17].

With predecitive models to classify programmers, new tools and IDEs[1] to teach programming can be developed. Such tools would provide different hints to programmers depending on their level of expertise. A novice Java programmer could be instructed to use inheritance and polymorphism; for average developers, functional idioms using lambda expressions could be introduced [94]; and advanced patterns to avoid performance bottlenecks or security vulnerabilities could be advised to expert programmers [95].

A system capable of classifying programmers by their expertise level can also be used to analyze the recurrent idioms written by expert programmers. Such idioms could be published and used to improve the skills of average programmers. Likewise, programming lecturers can identify the recurrent programming patterns used by beginners, explaining how they could be improved with better alternatives.

A model that scores the expertise level of programmers can be used to check the improvement of student's programing skills during a programming course. The model would identify those students that do not obtain the expected level of programming expertise, so lecturers could help them at the earliest. It would also identify those who have better programming skills, so they could be motivated with additional activities.

The scoring model could also be used by an Intelligent Tutoring System (ITS) [20] that considers how the student evolves. If the student score increases, more advanced programming constructs will be taught. If the score stays the same, the ITS will offer new activities to strengthen the new language construct taught. Finally, if the score drops, the system will revisit some language constructs formerly explained. In this case, the language construct to be revisited

[1] The classifier could also be included in existing IDEs such as Eclipse, IntelliJ and NetBeans.

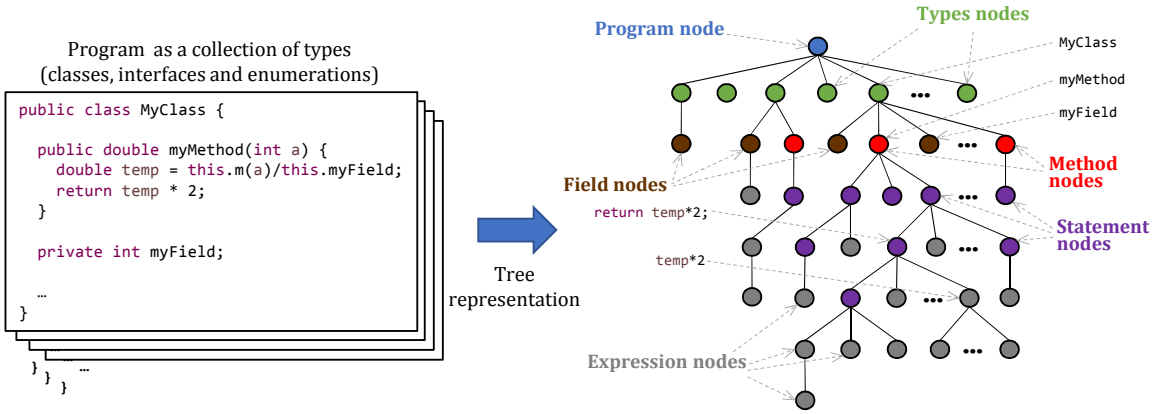


Figure 6.1: Program representation as heterogeneous compound syntax trees.

would depend on the idioms coded by the student.

6.1 Requirements

We face the challenge of building syntactic models to classify and score the programming level of expertise of Java developers. What follows are the main requirements we must fulfill.

6.1.1 Different levels of syntax constructs

When classifying programmers, different fragments of their code could be analyzed. Therefore, a classifier must consider different levels of syntax constructs, such as expressions, statements, methods, fields, types (classes, interfaces and enumerations) and whole programs (Figure 6.1). A whole program will give the classifier more information to label the developer, but a useful tool should give hints to the programmer when one single statement, method or even expression is typed. Therefore, a programmer classifier should be constructed with different models that classify different levels of syntax constructs (expressions, statements, methods, fields, types and whole programs).

6.1.2 Heterogeneous compound structures

Figure 6.1 shows that the syntax of the different language constructs are heterogeneous. For example, the syntax of methods is different to the syntax of statements and expressions. Moreover, many program constructs are composed of other program constructs. For example, the assignment statement "`temp = this.m(a)/this.myField;`" comprises the two expressions on the left- and right-hand side of the assignment operator (the right-hand side is also subdivided in other subexpressions). Likewise, object-oriented programs are composed of a set of types, types may comprise methods and fields, methods contain statements, and statements and fields are commonly built using expressions. Therefore, a Java syntax classifier should be able to label those heterogeneous compound AST structures.

6.1.3 Interpretable white-box models

As mentioned, the syntax patterns used to classify expert and novice programmer are valuable information. We described how they could be used to assist lecturers in a programming course, and to create Intelligent Tutoring Systems. Additionally, we propose to use the extracted patterns in a feature learning process (Section 6.3) to build classifiers with different kinds of syntax constructs.

6.1.4 Scalability

Model construction must be scalable, since we follow the big code philosophy of using massive codebases. It must allow the construction of classifiers from millions of instances. For example, just the dataset we used to build the expressions classifier holds 13,498,005 instances (see Section 6.4.1).

6.1.5 Models from trees

An important challenge of syntax pattern classification is to build predictive models from trees, since most supervised classification algorithms require instances (individuals or rows) to be represented as fixed size n-dimensional vectors [49]. While there are standard techniques to compute such vectors for documents, images and sound, there are no similarly standard representations for programs [15]. There exist alternative structured prediction methods such as Graph Neural Networks (GNNs) and Conditional Random Fields (CRFs)—discussed in Section 2.6—, but they unfortunately seem to suffer sufficiently high computation and space costs to be used with massive codebases [96].

6.2 Objective

We use decision trees (DTs) as the supervised learning algorithm, because DTs create interpretable white-box models, and perform well with large datasets [97]. They are also able to handle both numerical and categorical data.

In order to build DTs, we tabularize the ASTs of the input programs. We represent as features the main syntactic characteristics of each kind of node (expression, statement, method, field, type and program), including its category (e.g., arithmetic operation, method invocation, field access, etc.), and multiple information about their context (data about its parent and child nodes, its role in the enclosing node, its depth and height, etc.).

We create different datasets for each kind of node. Then, we build different homogeneous DT models that classify each kind of syntax construct (e.g., expressions, statements, methods, etc.). Finally, we take the patterns discovered by the homogeneous models to build new classifiers of compound heterogeneous syntax constructs (e.g., a method classifier that also considers the syntax patterns of the statements and expressions written within the method).

The main objectives of this ProgQuery use case scenario are:

1. A new feature learning approach to classify great amounts of compound heterogeneous tree structures.
2. A system to classify the programming expertise level of Java developers by analyzing the syntax constructs of their code. The system can also be used to measure the probability of a code fragment to be written by an expert or beginner.
3. The identification of Java syntax patterns used by both expert and novice programmers.

6.3 Methodology

Figure 6.2 shows the architecture of our system, and Algorithm 1 details how it works. We first provide a brief high-level description of the modules in the architecture. Forthcoming subsections detail the behavior of each module.

The input of the system is a database of labeled Java programs (expert or beginner); the output is a collection of heterogeneous decision tree models to classify programmers, plus the syntax patterns used by the classifier. Such patterns describe common idioms used by experts and beginners.

First, we use ProgQuery to create six homogeneous datasets with different features for each syntax construct: expressions, statements, methods, fields, types and programs. For each node in the ASTs, we store its features in a homogeneous dataset defined for that syntax construct. Such features include its syntactic category and context information (see Section 6.3.1). Then, for each syntax construct, we create a DT model capable of classifying all the different types of AST nodes defined for that syntax construct. For example, the expression DT model classifies any arithmetic, comparison, logical, variable, literals and cast expressions.

The next step is to obtain the syntax patterns from the homogeneous DT models (Section 6.3.2). DTs are traversed to obtain the decision rules used by the classifiers. The homogeneous syntax patterns are the antecedent parts of such decision rules. Those patterns are then reduced in number, and simplified to make them more readable (Section 6.3.3).

Once the homogeneous syntax patterns are reduced in number and simplified, we create the heterogeneous models (Section 6.3.4). As mentioned, AST structures are heterogeneous (e.g., programs, types, methods, fields, etc.) and some of them comprise other ones (e.g., programs contain types, and types contain methods and fields). Therefore, each heterogeneous dataset for one syntax construct (e.g., program) is built with its homogeneous dataset, plus all the syntax patterns of the subASTs it may contain (e.g., methods, fields, etc.). In this way,

[2]The pseudocode in Algorithm 1 uses the following functions: *classificationRules*, described in Section 6.3.2; *select* and *simplify*, depicted in Section 6.3.3; *potentialChildNodesOf*, which returns the syntax constructs that may occur as subtrees of another given syntax construct; and *columns*, which returns the syntax pattern columns in a given dataset.

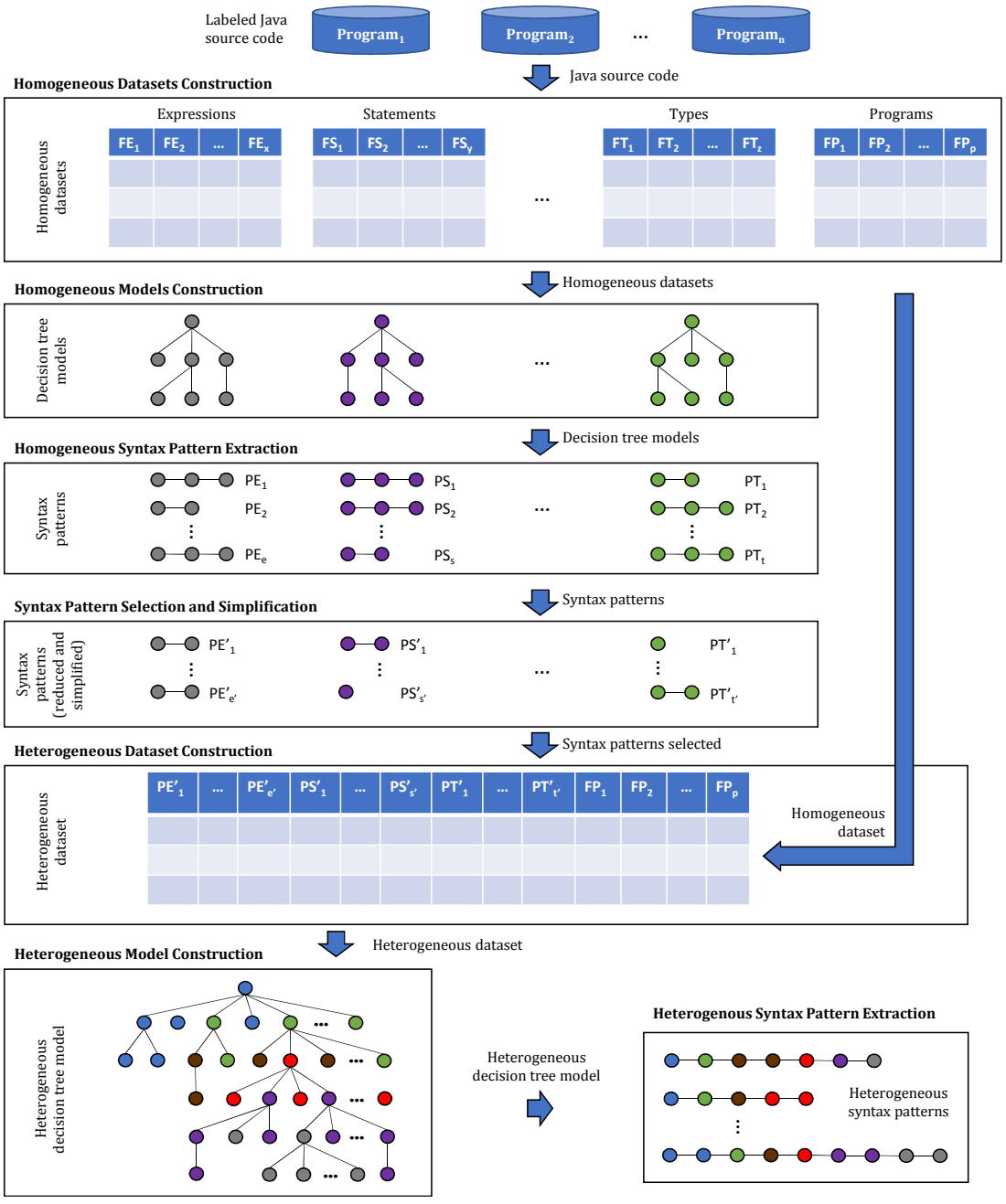


Figure 6.2: Architecture of the feature learning process.

Input : *sourceCodeDB*: collection of labeled Java programs.
Output: *heteroDT*: decision tree models, *heteroPatterns*: syntax patterns.

```
// Homogeneous model construction
foreach syntaxConstr in {expression, statement, method, field, type, program} do
    // Homogeneous dataset construction
    Define the structure of homoDS_syntaxConstr dataset, including its syntactic category
      and information about its context
    foreach AST in sourceCodeDB do
        foreach node of type syntaxConstr in AST do
            Include the node features and context information as a record (instance) in
              homoDS_syntaxConstr dataset
        end
    end
    // Homogeneous model construction
    Build a homoDT_syntaxConstr DT model using the homoDS_syntaxConstr dataset
    // Homogeneous syntax pattern extraction
    homoPatterns_syntaxConstr ← classificationRules(homoDT_syntaxConstr)
    // Syntax pattern selection and simplification
    homoPatterns_syntaxConstr ← select(homoPatterns_syntaxConstr)
    homoPatterns_syntaxConstr ← simplify(homoPatterns_syntaxConstr)
end
// Heterogeneous model construction
foreach syntaxConstr in {expression, statement, method, field, type, program} do
    // Heterogeneous dataset construction
    Define the structure of heteroDS_syntaxConstr dataset, including all the features
      (columns) of homoDS_syntaxConstr
    foreach childSyntaxConstr in potentialChildNodeOf(syntaxConstr) do
        Add to the structure of heteroDS_syntaxConstr dataset one feature (column) per
          syntax pattern in homoPatterns_childSyntaxConstr
    end
    Copy all the values from homoDS_syntaxConstr dataset to heteroDS_syntaxConstr
    foreach instance in heteroDS_syntaxConstr do
        foreach syntaxPattern in columns(heteroDS_syntaxConstr) do
            Update the (instance, syntaxPattern) cell in heteroDS_syntaxConstr with the
              percentage of occurrences of syntaxPattern in the subASTs of the AST
              represented by instance
        end
    end
    // Heterogeneous model construction
    Build a heteroDT_syntaxConstr DT model using the heteroDS_syntaxConstr dataset
    // Heterogeneous syntax pattern extraction
    heteroPatterns_syntaxConstr ← classificationRules(heteroDT_syntaxConstr)
end
```

Algorithm 1: Pseudocode describing the proposed method[2].

Production		Syntactic category
expression	→ expression (‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’) expression	Arithmetic (binary)
	| expression (‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘==’ | ‘!=’) expression	Comparison
	| expression (‘&&’ | ‘||’) expression	Logic (binary)
	| expression (‘&’ | ‘|’ | ‘^’) expression	Bitwise (binary)
	| expression ‘instanceof’ type	Instance of
	| expression (‘=’ | ‘+=’ | ‘-=’ | . . .) expression	Assignment
	| expression (‘>>’ | ‘<<’ | ‘>>>’) expression	Shift
	| expression ‘?’ expression ‘:’ expression	Conditional
	| (‘++’ | ‘--’) expression	Inc-Dec prefix
	| expression (‘++’ | ‘--’)	Inc-Dec postfix
	| (‘+’ | ‘-’) expression	Arithmetic (unary)
	| ‘!’ expression	Logic (unary)
	| ‘∼’ expression	Bitwise (unary)
	| ‘(’ type ‘)’ expression	Cast
	| expression ‘.’ ID	Field access
	| expression ‘::’ ID	Method reference
	| expression ‘[’ expression ‘]’	Indexing
	| expression ‘(’ (expression (‘,’ expression)*)? ‘)’	Invocation
	| lambda_parameters ‘->’ lambda_body	Lambda
	| ‘new’ type ‘(’ (expression (‘,’ expression)*)? ‘)’	New object
	| ‘new’ type (‘[’ expression ‘]’)+ (‘[’ ‘]’)*	New array
	| ID	Identifier
	| INT_LITERAL | CHAR_LITERAL | . . .	Int, char literal . . .

Figure 6.3: Feature abstraction function for the syntactic category of expressions.

heterogeneous classifiers learn from not only their homogeneous features, but also the most relevant syntax patterns of their subASTs.

The last step of the process involves extracting the heterogeneous syntax patterns from the heterogeneous DT models (Section 6.3.5), the same way as we did with the homogeneous ones. The extracted patterns represent common Java idioms used by experts and beginners.

6.3.1 Homogeneous datasets and models construction

As mentioned, classical supervised learning algorithms work on *feature vectors*: n-dimensional vectors of features that represent each instance (i.e., each individual in a given problem). Our approach is to translate homogeneous syntax constructs (expressions, statements, etc.) into vectors of features. For each syntax construct, we define a *feature abstraction* set of functions f that map each AST node to a numeric value encoding one property [49]. Given the set of feature abstractions $f_1, \ldots, f_n$, we can represent a given AST t as the feature vector $[f_1(t), \ldots, f_n(t)]$. In this way, feature abstraction functions represent a parameterizable mechanism to translate ASTs as feature vectors.

The first feature abstraction function we define is the syntactic category of each node in the AST. Figure 6.3 shows the syntactic category feature abstraction for Java expressions (we use the ANTLR meta-language notation [98]). This feature simply denotes the kind of expression an AST node represents (its syntactic category).

Name	Description
Category	Syntactic category of the current node, detailed in Figure 6.3.
First, second and third child	Syntactic category of the corresponding child node.
Parent node	Syntactic category of the parent node.
Role	Role played by the current node in the structure of its parent node.
Height	Distance (number of edges) from the current node to the root node in the enclosing type (class, interface or enumeration).
Depth	Maximum distance (number of edges) of the longest path from the current node to a leaf node.

Table 6.1: Feature abstractions used for expressions.

Besides its syntactic category, we also use ProgQuery to gather context information of AST nodes. Each AST node occurs in some surrounding context (e.g., parent and child nodes), and we want the classifier to make decisions based on such contexts. For example, object construction using the `new` operator does not discriminate the level of expertise. However, our system detects that beginners rarely use such expression to initialize a `final` field in a class. Thus, the context of a non-discriminating expression may be discriminating.

Table 6.1 includes the context information stored for expressions (the features used for the rest of syntax constructs can be consulted in Appendix C). We use feature abstraction functions to represent the syntactic categories of its three potential child nodes (if no child exits, e.g. unary expressions, the feature is assigned zero), together with its parent. We also store the role that the expression is playing in the parent node. For example, if the parent is the conditional ternary operator, the current node could be playing the role of the condition (first child node), the if-true expression (second child node) or the if-false expression (third child node). As shown in Table 6.1, we also store the node depth and height in the AST.

To fill the datasets with the syntactic information taken from the labeled source code, we used the ProgQuery Java compiler plug-in (Section 3.2)[1]. We get the syntactic representation of Java programs, and convert the AST structures into the tabular information defined for each syntax construct. Once the homogeneous datasets for the different syntax constructs are built, we create the homogeneous DT models.

6.3.2 Homogeneous syntax pattern extraction

We get the classification rules from the homogeneous decision trees. DTs are traversed with an instance of the *Visitor* design pattern [33], storing the paths from root to leaf nodes as classification rules. The antecedents of the classifica-

[1]Cypher could have been used to extract the syntactic information with ProgQuery. However, we started the program classification project before ProgQuery was finished.

tion rules represent syntax patterns, and the consequent is the outcome of the classification. For example, one decision rule for expressions is:

```
if category(node) == assignment
    and category(first_child(node)) == field_access
    and category(second_child(node)) == new_object
    and role_in_parent(node) == expression
then expertise_level = expert
```

This rule classifies the example expression "`obj1.m(obj2.f = new MyClass())`" as code written by an experimented programmer. For this rule, the syntax pattern gathered is an assignment that plays the role of an expression (instead of statement), its left-hand side expression is a field access, and the right-hand side is an object construction.

6.3.3 Syntax pattern selection and simplification

The heterogeneous datasets include one feature per homogeneous syntax pattern of their potential subASTs. This process would produce a huge number of features, since we expect the number of homogenous patterns to be high. Moreover, the compound nature of ASTs make the number of features to be even higher. For example, the heterogeneous features for programs include the syntax patterns of types, methods, fields, statements and expressions.

Our intention is thus to reduce the number of patterns with the minimal reduction of the accuracy of the classifier, finding a trade-off between these two conflicting variables. To this end, we consider two measures of syntax patterns: coverage and confidence [99]. Coverage is defined as the relative number of instances that satisfy the pattern (how frequently the pattern appears in the dataset):

$$Coverage(pattern) = \frac{occurrences\ of\ pattern\ in\ dataset}{number\ of\ instances} \tag{6.1}$$

Confidence is a measure for the whole classification rule, not just the antecedent. The confidence of a rule is an indication of how often a rule has been found to be true:

$$Confidence(rule) = \frac{instances\ fulfilling\ the\ rule}{occurrences\ of\ rule\ antecedent\ in\ dataset} \tag{6.2}$$

We analyze the influence of these two measures on the accuracy of the classifier. Then, we discard all those rules that do not involve a significant reduction in the classifier performance (experimental results are presented in Section 6.4.3). We also perform some rule simplifications to make syntax patterns more readable.

6.3.4 Heterogeneous dataset and model construction

The previous pattern extraction and selection processes undertake automatic feature learning to build the final heterogeneous compound classifiers. The dataset of each syntax construct is made up of the selected syntax patterns of their subASTs (Section 6.3.3), together with the features of the corresponding homogeneous dataset (Section 6.3.1)—see Figure 6.2 and Algorithm 1. The outcome is a collection of different datasets that are used to build the final heterogeneous compound models.

In the heterogeneous datasets, we set the value of each syntax pattern feature to the percentage of occurrence of such patterns. For each compound instance (e.g., one statement), we count the syntax patterns that its child nodes (its subexpressions) fulfill. Then, the cells for such syntax patterns are filled in with the percentage of occurrence of the pattern in the AST represented by that instance (statement). In this way, the homogeneous features of one syntax construct are enriched with the syntax patterns of its child subASTs, providing better classification performance. In Figure 6.2, the homogeneous program features $(FP_1, FP_2, \ldots, FP_p)$ are enriched with all the heterogeneous syntax patterns that could be found in the program subASTs: types $(PT'_1, \ldots, PT'_{t'})$, methods, fields, statements $(PS'_1, \ldots, PS'_{s'})$ and expressions $(PE'_1, \ldots, PE'_{e'})$.

From the implementation point of view, much computation time is required to check whether subtrees in a program fulfill a specific syntax pattern. To optimize this operation, we convert all the syntax patterns to SQL queries against the homogeneous datasets in the database. The premises in the pattern are translated to SQL "`where`" clauses. Those queries are executed programmatically, and their results are used to fill in the heterogeneous datasets.

6.3.5 Heterogeneous syntax pattern extraction

As with the homogeneous datasets, we traverse the resulting heterogeneous DTs to obtain the final heterogeneous syntax patterns. Each classification rule obtained represents a compound syntax pattern for a given programming expertise level. Such rules are expressed not only with features of the syntax construct to be classified, but also with syntax patterns for its child nodes.

6.4 Evaluation

In this section, we evaluate the performance of the proposed system to label Java programmers according to their expertise level. We first describe the experimental data (Section 6.4.1) and environment (Section 6.4.2). Then, we describe and show the results of the following experiments within the framework of the proposed system:

1. Syntax pattern selection (Section 6.4.3). This experiment applies the method for pattern selection described in Section 6.3.3 to reduce the number of syntax patterns taken from different DTs.

2. Heterogeneous AST classification (Section 6.4.4). Evaluates the accuracy of DTs to classify heterogeneous ASTs. DTs are compared with the existing related work, and other common machine-learning approaches.

3. Heterogeneous syntax pattern extraction (Section 6.4.5). We analyze the final syntax patterns obtained by our system.

4. Scoring the expertise level of programmers (Section 6.4.6). The heterogeneous dataset is used to score the programming expertise level of Java developers.

5. Execution time of the proposed method (Section 6.4.7). We measure the execution time required for each module of the architecture described in Section 6.3.

6.4.1 Experimental data

To build the datasets, we took Java code from different sources and labeled them as either beginner or expert programmer. For beginners, we gathered code from first year undergraduate students in a Software Engineering degree at the University of Oviedo. We took the code they wrote for the assignments in two year-1 programming courses in academic years 2017/18 and 2018/19. All the code was 100% written by students from scratch. Overall, we collected 35,309 Java files from 3,884 programs.

For expert programmers, we took the source code of different public Java projects in GitHub. We selected active projects with the highest number of contributors: Chromium, LibreOffice, MySQL, OpenJDK and Amazon Web Services. These software products are implemented with 43,775 Java files in 137 programs (AWS comprises 133 different projects).

We used ProgQuery to get the syntactic information (features in the homogeneous dataset) from the Java code. We gather all the values for the features of the different syntax constructs, filling in the values in the homogeneous datasets. Then, we label each instance with its expertise level.

Table 6.2 shows the number of AST nodes. To balance data, we randomly removed instances from the over-representing class to get the same exact number of instances for both classes (final column in Table 6.2).

6.4.2 Experimental environment

To implement the experiments, we used Python 3.7.2 and scikit-learn 0.21.1. All the datasets were stored in a PostgreSQL database 11.3. Since PostgreSQL limits the maximum number of columns to 1600, we modified its open source implementation to allow 12,800 columns (features). We run all the code in a Dell PowerEdge R530 server with two Intel Xeon E5-2620 v4 2.1GHz microprocessors (32 cores) with 128GB DDR4 2400MHz RAM memory, running CentOS operating system 7.4-1708 for 64 bits.

	Beginner	Expert	Total	Final
Expressions	4,616,807	8,881,198	13,498,005	9,233,614
Statements	1,304,585	2,292,791	3,597,376	2,609,170
Methods	237,285	370,618	607,903	474,570
Fields	96,175	135,826	232,001	192,350
Types	35,910	58,719	94,629	71,820
Programs	3,884	137	4,021	274
Total	6,294,646	11,739,289	18,033,935	12,581,798

Table 6.2: Number of AST nodes.

DTs were created with the CART algorithm implemented by scikit-learn (`DecisionTreeClassifier`) [100]. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node. This implementation of decision trees permits the use both categorical and numerical data. We selected the best hyper-parameters with exhaustive parallel search across common parameter values (`GridSearchCV`), using stratified randomized 10-fold cross validation (`StratifiedShuffleSplit`). For the hyper-parameter to measure the quality of a split, we tried `gini` and `entropy`; for selecting the strategy to choose the split at each node, we tried `best` and `random`.

6.4.3 Syntax pattern selection

With our source code database of 35,309 Java files, we generated the six homogeneous datasets detailed in Table 6.2. On aggregate, the datasets contain 12.5 million AST nodes (instances). The six homogeneous DT models were created, and syntax patterns were extracted as explained in Section 6.3.2. That pattern extraction process produced 45,590 patterns for all the homogeneous models (10,562 for expressions; 28,163 for statements; 4,806 for methods; 336 for fields; 1,702 for types; and 21 for programs). Since this number poses high dimensionality to build a predictive model [101], we define the mechanism to reduce the number of syntax patterns described in Section 6.3.3.

We want to reduce the number of syntax patterns with the minimal reduction of classifier accuracy. To this aim, we analyze the influence of rule (syntax pattern) coverage, confidence, precision and recall on the accuracy of the whole DT classifier. Figure 6.4 presents the results. For each measure, it shows the accuracy of the classifier (y-axis) built with $n\%$ of the rules (x-axis) with the highest coverage, confidence, precision and recall. Figure 6.4 shows how, for all the datasets, coverage was the measure that selected the lowest number of patterns with the highest accuracy of the classifier.

Sorting the classification rules by coverage, we have to choose a percentage of rules (preferably low) with little penalty on the classifier accuracy. To this end, we used the Coefficient of Variation (CoV), defined as the ratio of the standard deviation to the mean. We measured the CoV of the classifier accuracy for the last ten percentages of rules in Figure 6.4, and selected the first percentage of rules where such CoV is lower than 2%. As shown in Figure 6.4, that value

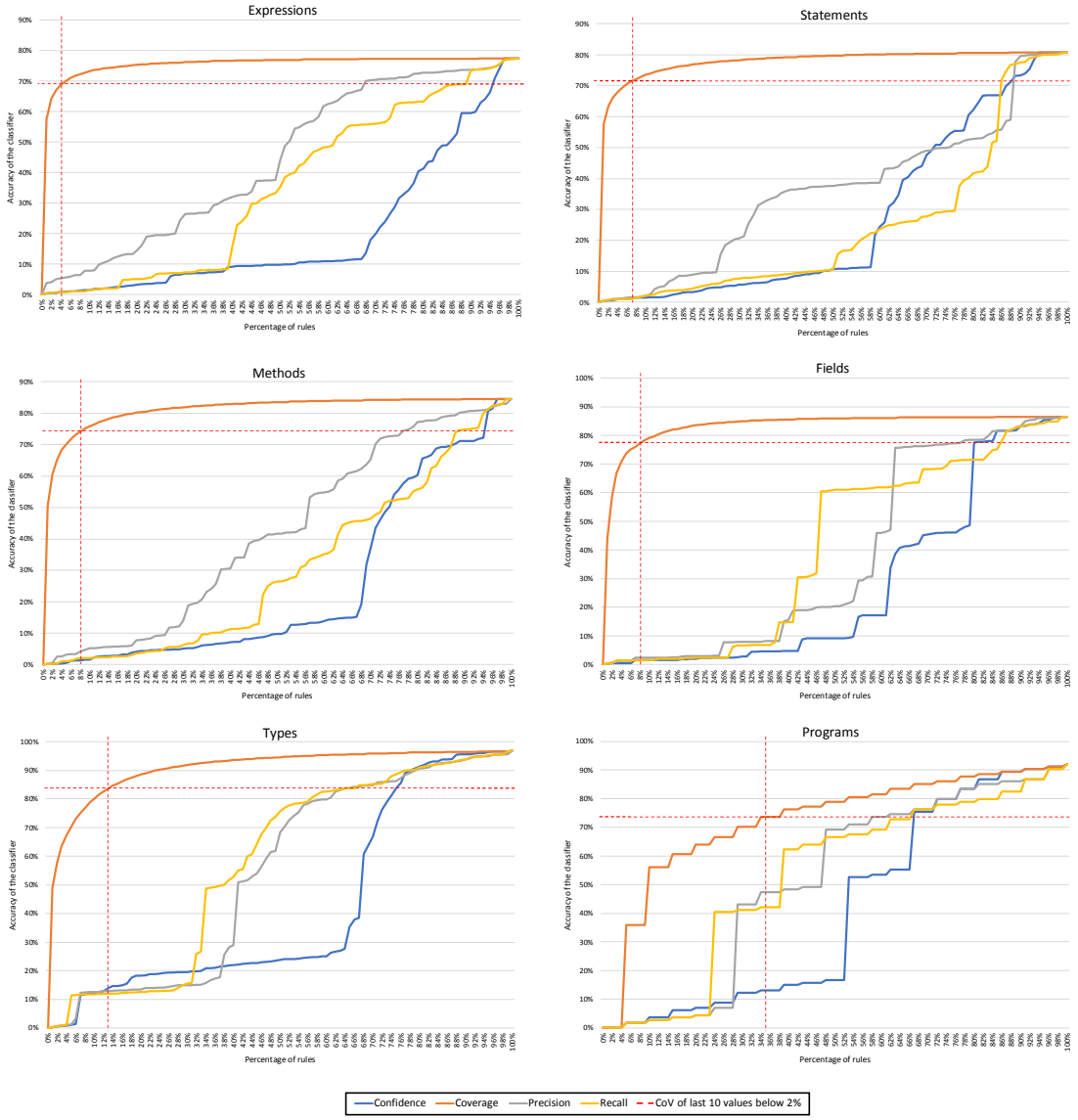


Figure 6.4: Classifier accuracy (y-axis) obtained with a percentage of rules (x-axis) with the highest confidence, coverage, precision and recall. For confidence, it is shown the CoV of the last 10 values below 2%.

approximates the elbow value in all the coverage curves, representing a good trade-off between syntax pattern pruning and classifier accuracy.

Table 6.3 details the results of pattern selection. Out of 45.569 rules (syntax patterns), we select 3.027 (6.6%). Moreover, the average accuracy of the model is reduced in just 12.4% (from 10.6% in fields to 16.2% in programs).

6.4.4 Heterogeneous AST classification

Section 6.3.4 describes how heterogeneous datasets are created by combining the homogeneous datasets with the syntax patterns selected in the previous experiment. Now, we evaluate the performance of the heterogeneous DT models. For that purpose, we divide all the datasets into 80% of the instances for training and 20% for testing, using a stratified random sampling method [102]. We repeat the

	Original rules	Rules selected	Rule reduction	Accuracy loss
Expressions	10,562	422	96,0%	11,1%
Statements	28,163	1,971	93,0%	11,6%
Methods	4,806	384	92,0%	12,3%
Fields	336	27	92,0%	10,6%
Types	1,702	221	87,0%	13,5%
Programs	21	7	65,0%	16,2%
Total	45,590	3,034	93.3%	12.4%

Table 6.3: Results of pattern selection.

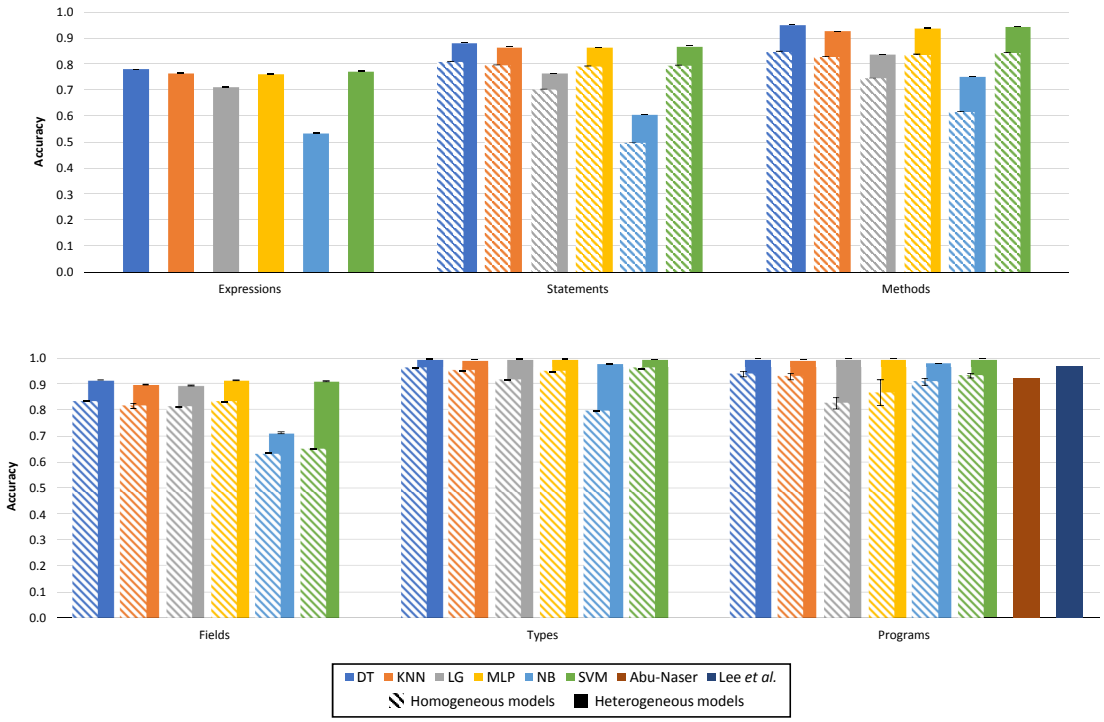


Figure 6.5: Accuracy of all the classifiers (whiskers represent 95% confidence intervals).

training plus testing process 30 times, measuring the mean, standard deviation and 95% confidence intervals of accuracy, F1 and AUC values [102]. Data split was random and stratified to ensure that the proportions between classes are the same in each fold, as they are in the whole dataset (50% / 50%).

In Figure 6.5, we can see the accuracy of the heterogeneous DT models. When a whole program is classified, the average accuracy of DTs is 99.6%. This performance is reduced when we classify smaller AST structures: 99.5% for types, 95.2% for methods, 91.4% for fields, 88.3% for statements, and 78.1% for expressions. Variability (confidence intervals) of the results is low (Table 6.4), because models were created with an important amount of data (more than 12 million instances in total; see Table 6.2). 95% confidence intervals for all the models are lower than 0.04% (Table 6.4).

Figure 6.5 compares our system with the two related works to classify the expertise level of programmers, discussed in Section 2.3. The research work un-

dertaken by Abu-Naser and Lee *et al.* classifies programmers with 92% and 97% accuracy, respectively. Our system provides 99.6% average accuracy with 0.02% error, so there seem to be a significant benefit[1]. Moreover, our system predicts the expertise level of programmers by just analyzing their code; it does not need to interact or observe them while they are coding.

		DT	KNN	LG	MLP	NB	SVM
Accuracy	Statements	**0.883** ± 0.04%	0.866 ± 0.04%	0.765 ± 0.03%	0.864 ± 0.03%	0.605 ± 0.06%	0.869 ± 0.04%
	Methods	**0.952** ± 0.03%	0.927 ± 0.03%	0.837 ± 0.02%	0.938 ± 0.03%	0.752 ± 0.05%	0.946 ± 0.04%
	Fields	**0.914** ± 0.03%	0.896 ± 0.03%	0.891 ± 0.04%	0.912 ± 0.04%	0.711 ± 0.06%	0.910 ± 0.04%
	Types	**0.995** ± 0.02%	0.992 ± 0.02%	**0.995** ± 0.02%	**0.995** ± 0.01%	0.978 ± 0.04%	0.993 ± 0.02%
	Programs	**0.996** ± 0.02%	0.992 ± 0.03%	**0.996** ± 0.02%	**0.996** ± 0.02%	0.980 ± 0.04%	**0.996** ± 0.01%
F1	Statements	**0.883** ± 0.03%	0.866 ± 0.04%	0.765 ± 0.03%	0.864 ± 0.03%	0.606 ± 0.05%	0.870 ± 0.04%
	Methods	**0.952** ± 0.03%	0.927 ± 0.03%	0.837 ± 0.02%	0.938 ± 0.03%	0.753 ± 0.05%	0.947 ± 0.04%
	Fields	**0.914** ± 0.02%	0.896 ± 0.03%	0.891 ± 0.04%	0.912 ± 0.04%	0.711 ± 0.06%	0.910 ± 0.04%
	Types	**0.995** ± 0.02%	0.992 ± 0.02%	**0.995** ± 0.02%	**0.995** ± 0.01%	0.979 ± 0.04%	0.993 ± 0.02%
	Programs	**0.996** ± 0.02%	0.992 ± 0.03%	**0.996** ± 0.02%	**0.996** ± 0.02%	0.980 ± 0.04%	**0.996** ± 0.01%
AUC	Statements	**0.883** ± 0.04%	0.866 ± 0.04%	0.765 ± 0.03%	0.864 ± 0.03%	0.605 ± 0.06%	0.869 ± 0.04%
	Methods	**0.952** ± 0.03%	0.927 ± 0.03%	0.837 ± 0.02%	0.938 ± 0.03%	0.752 ± 0.05%	0.946 ± 0.04%
	Fields	**0.914** ± 0.03%	0.896 ± 0.03%	0.891 ± 0.04%	0.912 ± 0.04%	0.711 ± 0.06%	0.910 ± 0.04%
	Types	**0.995** ± 0.02%	0.992 ± 0.02%	**0.995** ± 0.02%	**0.995** ± 0.01%	0.978 ± 0.04%	0.993 ± 0.02%
	Programs	**0.996** ± 0.02%	0.992 ± 0.03%	**0.996** ± 0.02%	**0.996** ± 0.02%	0.980 ± 0.04%	**0.996** ± 0.01%

Table 6.4: Performance of all the heterogeneous models (95% confidence intervals are expressed as percentages). Bold font represents the highest value. If one row has multiple cells in bold type, it means that there is not significant difference among them (p-value ≥ 0.05, $\alpha = 0.05$).

		DT	KNN	LG	MLP	NB	SVM
Accuracy	Expressions	**0.781** ± 0.01%	0.766 ± 0.17%	0.714 ± 0.01%	0.762 ± 0.15%	0.534 ± 0.01%	0.773 ± 0.16%
	Statements	**0.810** ± 0.02%	0.795 ± 0.16%	0.701 ± 0.03%	0.793 ± 0.02%	0.496 ± 0.03%	0.797 ± 0.14%
	Methods	**0.850** ± 0.04%	0.828 ± 0.06%	0.748 ± 0.06%	0.837 ± 0.05%	0.616 ± 0.06%	0.844 ± 0.05%
	Fields	**0.834** ± 0.08%	0.818 ± 1.31%	0.814 ± 0.08%	0.833 ± 0.07%	0.629 ± 0.11%	0.651 ± 0.10%
	Types	**0.962** ± 0.05%	0.954 ± 0.09%	0.916 ± 0.10%	0.951 ± 0.10%	0.798 ± 0.12%	**0.962** ± 0.07%
	Programs	**0.940** ± 1.18%	0.932 ± 1.36%	0.827 ± 2.53%	0.868 ± 5.68%	0.910 ± 1.42%	0.932 ± 1.02%
F1	Expressions	**0.780** ± 0.01%	0.771 ± 0.16%	0.709 ± 0.01%	0.770 ± 0.13%	0.139 ± 0.13%	0.773 ± 0.16%
	Statements	0.801 ± 0.02%	0.804 ± 0.15%	0.682 ± 0.03%	0.786 ± 0.03%	0.631 ± 0.02%	**0.807** ± 0.13%
	Methods	**0.843** ± 0.05%	0.842 ± 0.05%	0.732 ± 0.08%	0.829 ± 0.06%	0.431 ± 0.18%	0.831 ± 0.06%
	Fields	**0.837** ± 0.07%	0.818 ± 1.61%	0.816 ± 0.08%	0.836 ± 0.06%	0.431 ± 0.60%	0.738 ± 0.05%
	Types	**0.961** ± 0.06%	0.953 ± 0.09%	0.914 ± 0.10%	0.949 ± 0.11%	0.766 ± 0.16%	**0.961** ± 0.07%
	Programs	**0.940** ± 1.17%	0.930 ± 1.42%	0.839 ± 2.16%	0.869 ± 6.45%	0.912 ± 1.39%	0.931 ± 1.02%
AUC	Expressions	**0.781** ± 0.01%	0.766 ± 0.17%	0.714 ± 0.01%	0.762 ± 0.15%	0.534 ± 0.01%	0.773 ± 0.16%
	Statements	**0.810** ± 0.02%	0.795 ± 0.16%	0.701 ± 0.03%	0.793 ± 0.02%	0.496 ± 0.03%	0.797 ± 0.14%
	Methods	**0.850** ± 0.04%	0.828 ± 0.06%	0.748 ± 0.06%	0.837 ± 0.05%	0.616 ± 0.06%	0.844 ± 0.05%
	Fields	**0.834** ± 0.08%	0.818 ± 1.31%	0.814 ± 0.08%	0.833 ± 0.07%	0.629 ± 0.11%	0.651 ± 0.10%
	Types	**0.962** ± 0.05%	0.954 ± 0.09%	0.916 ± 0.10%	0.951 ± 0.10%	0.798 ± 0.12%	**0.962** ± 0.07%
	Programs	**0.940** ± 1.18%	0.932 ± 1.35%	0.827 ± 2.52%	0.868 ± 5.72%	0.910 ± 1.40%	0.932 ± 1.03%

Table 6.5: Performance of all the homogeneous models (95% confidence intervals are expressed as percentages). Bold font represents the highest value. If one row has multiple cells in bold type, it means that there is not significant difference among them (p-value ≥ 0.05, $\alpha = 0.05$).

Figure 6.5 also presents the performances of the homogeneous DT models, comparing them with the heterogeneous ones. The purpose of this comparison is to see whether the addition of child subAST patterns actually increases

[1]The two related works do not provide 95% confidence intervals or data to compute a statistical hypothesis test. In addition, we could not repeat their experiments because they use electroencephalographic sensors, an eye-tracker, and a LP-ITS system that is not available for download.

the accuracy of the classifiers. All the heterogeneous DT models improve the performance of the corresponding homogeneous ones. Accuracies of statements, methods, fields, types and programs are increased in 9%, 12%, 9.6%, 3.4% and 6%, respectively.

Besides the DT models to classify programmers, we built other classifiers with distinct techniques, using the same datasets. As mentioned, we selected DTs because they are interpretable white-box models, which allow us to extract the syntax patterns of the classifiers in order to implement the feature learning approach proposed in this chapter. However, we want to see to what extent the classification accuracy of DTs is similar to other common machine-learning approaches.

In particular, we built other classifiers using logistic regression, (Gaussian) naïve Bayes, multilayer perceptron, support vector machines, and k-nearest neighbors. For all these models, we first perform a feature selection process and then hyper-parameter tuning. Features were selected with the `SelectFromModel` meta-transformer that chooses features depending on importance weights. The estimator used to select the features was a random forest classifier with 100 trees. Hyper-parameter tuning was done the same way as for DTs—the different hyper-parameter options used can be consulted in [103]. We repeat the training plus testing process 30 times, computing the 95% confidence intervals. All the algorithms, including feature selection and hyper-parameter tuning, were executed in parallel using all the cores in our server.

Figure 6.5 and Table 6.5 show the performance of the different classifiers for the homogeneous datasets. DT is the method with the best average performance. We performed a statistical hypothesis test ($\alpha = 0.05$) to compute whether the performance of each method is significantly different to DT ($p\text{-}value < 0.05$). For accuracy and AUC measures, DT provides the highest performance (in three cases, statistical differences with SVM are not significant; see Table 6.5). For F1, there is one case (statements) where SVM performs better than DT; in the rest of scenarios, DT provides the highest F1 measures. For the heterogeneous models (Table 6.4), DT is the technique with the highest performance for all the measures (accuracy, F1 and AUC). When classifying types and programs, LG, MLP and SVM are not significantly different to DT. All these results validate that DT not only builds interpretable white-box models, but also provide excellent performance results for the given datasets.

6.4.5 Heterogeneous syntax pattern extraction

The expert and novice syntax patterns found in the heterogeneous models are a valuable outcome of our research. As mentioned, they could be used in programming courses, to improve the hints given by development environments, and to create Intelligent Tutoring Systems. As described in Section 6.3.5, we traversed the heterogeneous DT models to extract the final syntax patterns used by expert and novice Java programmers. For example, the following rule classifies a programmer as expert, using syntax patterns of program, type and method constructs:

```
if enumeration_percentage(program)>0 and interface_percentage(program)>1
    and ∃ type1 . category(type1)==class and generic_types(type1)>0
    and ∃ type2 . category(type2)==class and extend_classes(type2)>0
        and implement_interfaces(type2)>1
    and ∃ method . number_statements(method)<=3
then expertise_level = expert
```

The previous pattern describes programs that contains enumeration and interface (more than 1%) types, implements no less than one generic type, at least one class extend another class and implements more than one interface, and one method has three or fewer statements.

Likewise, the following method pattern was extracted to classify beginners:

```
if not(isOverride(method)) and numberOfAnnotations(method)==0
    and numberOfParameters(method)==0 and not(isFinal(method))
    and numberOfThrows(method)==0 and numberOfStatements(method)<=2
    and visibility(method)==public and numberOfGenericTypes(method)==0
    and namingConvention(method)==snake_case
    and ∃ statement . depth(statement)>=67
then expertise_level = beginner
```

Our system extracted 742, 721, 782, 636 and 575 heterogeneous syntax patterns for, respectively, statements, methods, fields, types and programs. All the patterns found are available for download at [103].

6.4.6 Scoring the expertise level of programmers

An important discussion regarding the classification method proposed in this dissertation is about the binary character of classifying programmers as either experts or beginners. As we know, the classification of programmers by their expertise level is not binary, since many programmers may be classified as intermediate level. Fortunately, since the classifier infers the syntax patterns for novice and expert programmers, it is possible to measure the probability of being in one of these groups, and hence to score how close the programmer is to be an expert or beginner.

Logistic regression is a calibrated probabilistic classifier that provides a score that can be directly interpreted as a confidence level [104]. We built logistic regression models from the heterogeneous datasets to compute the probability of a programmer to be classified as novice or expert. Figure 6.6 shows the percentage of instances per score, for the two labels (beginner and expert). We can see how the most common score is a number between 0.9 and 1, because all the instances in the dataset are code written by either beginners or experts. As expected, expressions are the syntax patterns with the worst performance (71.4% of the instances are classified correctly), and programs outperform the rest of syntax

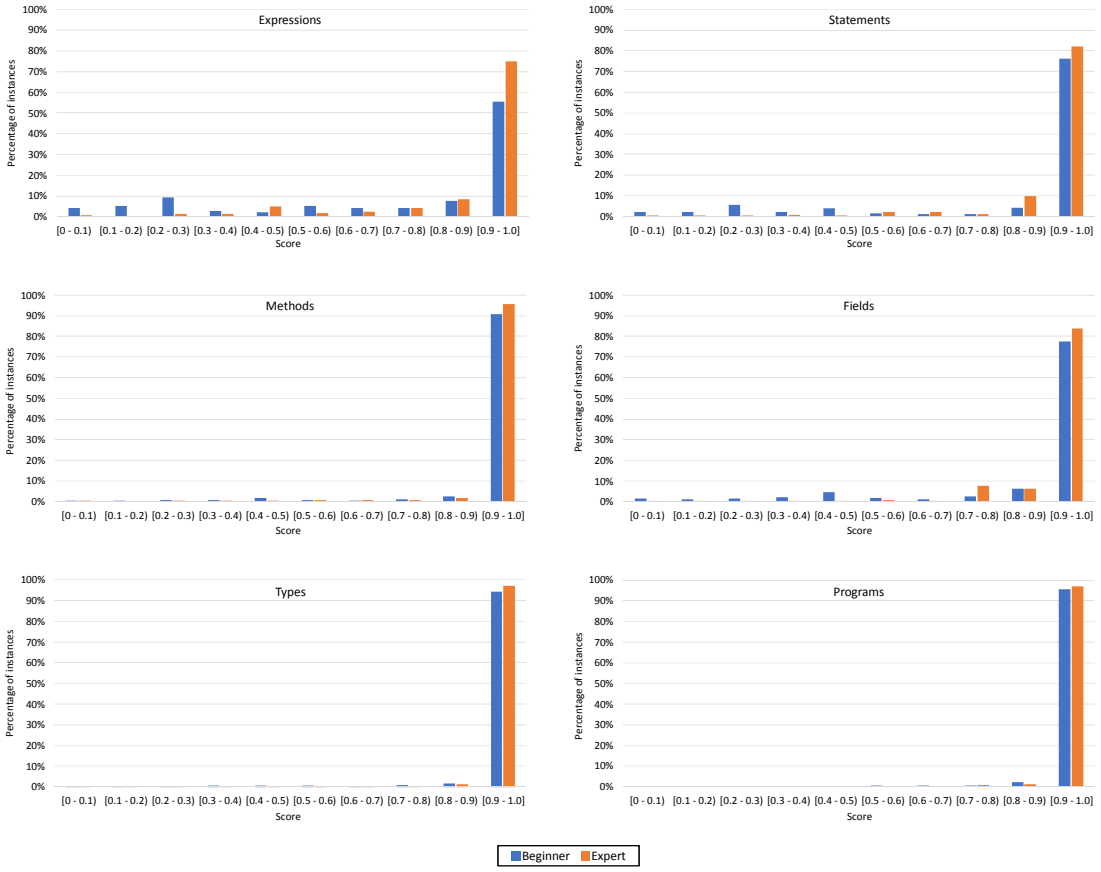


Figure 6.6: Percentage of instances per score, using a probabilistic LG model.

constructs (99.6%). We can also see in Figure 6.6 that the model classifies experts better than beginners. It seems to be easier to identify the syntax patterns that expert programmers write, rather than those coded by beginners.

6.4.7 Execution time of the proposed method

Table 6.6 shows the execution times of all the phases of the proposed system. The system takes as input the Java programs described in Section 6.4.1, and produces six heterogeneous classifiers plus the final syntax patterns for each syntax construct.

		Programs	Types	Methods	Fields	Statements	Expressions	Total
Homogen.	Dataset construction	1.961	46.14	296.4	113.1	1754	6582	8,793.2
	Model construction	0.030	0.193	0.064	0.028	0.487	2.375	3.2
	Syntax pattern extraction	0.009	1.043	13.24	0.439	312.5	1595	1,921.7
	Pattern selection and simplification	0.316	0.410	1.366	0.183	9.219	27.38	38.9
Heterogen.	Dataset construction	3320	3107	2399	1093	1168	—	11,087.6
	Model construction	0.210	0.241	0.209	0.117	0.654	—	1.4
	Syntax pattern extraction	0.914	1.164	21.21	0.994	401.1	—	425.4
	Pattern selection and simplification	0.449	0.534	2.456	0.436	11.954	—	15.8
	Total	3,324	3,156	2,734	1,208	3,658	8,206	22,287.3

Table 6.6: Execution times (seconds) of all the modules in the architecture.

The whole process took 6 hours and 12 minutes (22,287 seconds) to run in

the computer described in Section 6.4.2. The two most expensive phases are the construction of datasets, which take 2.44 and 3.08 hours for the homogeneous and heterogeneous datasets, respectively.

The generated DT classifiers are able to predict the expertise level of Java programmers almost instantaneously (average execution time is 4.6 microseconds). Logistic regression and multilayer perceptron perform similarly (differences are not statistically significant). Naïve Bayes, SVM and KNN require, respectively, 3.87, 987 and 6079 times more execution time than DT to classify a Java programmer.

Chapter 7

Conclusions

The proposed representation of source code by means of different overlapping graph structures can be used to build different tools for software development, such as efficient and scalable program analyses, and source code classifiers. This proposal is materialized with the implementation of ProgQuery, an infrastructure that builds those graph representations for Java code, and stores them in a Neo4j graph database. Syntactic and semantic information can be consulted in a declarative fashion, and its performance and scalability is higher than related systems.

We propose an ontology with seven graph structures to represent syntactic and semantic information of Java programs. This information includes Abstract Syntax Trees, Control Flow Graphs, Program Dependency Graphs, Call Graphs, Type Graphs, Class Dependency Graphs and Package Graphs. We represented different Java programs with these structures, and the information provided is sufficient to implement advanced well-known static analyses.

The seven representations defined are stored into a Neo4j graph database, so no impedance mismatch is produced; i.e., graphs are not translated into tables. One direct benefit is runtime performance, since no model translation is required. Another advantage is that graph abstractions are not mislaid in the persistence store, so any language, API and framework to access the database provide the original graph abstractions.

One distinguishing feature of the information stored is that graph representations are designed to be overlaid. This means that a syntactic node may be connected with other different semantic representations through semantic relationships, and vice versa. This makes it easy to express advanced analyses, because syntactic and semantic information can be combined. It also avoids the performance cost of connecting different representations when queries are executed.

The evaluation presented shows that ProgQuery runs static analyses from 48 to 245 times faster than the related systems. It has also shown better scalability to increasing program sizes and analysis complexity. ProgQuery is the only system that runs all the analyses for a 18M lines of code program, and most of the

analyses take tens of seconds. In ProgQuery, Cypher queries require from 18% to 50% the code needed to express the same analyses in other systems.

These performance and expressiveness benefits are partially caused by the additional semantic information provided by our system (the overlaid representations are also an important factor). 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 4,816% to 24,551%).

Besides static analysis, ProgQuery has been used to create a predictive model to classify Java code according to programmer's expertise. We extract homogeneous syntactic constructs and store them in different datasets. Then, a feature learning mechanism is used, combining homogeneous syntax patterns to build classifiers for heterogeneous compound tree structures. Classification performance ranges from 78.1% when labeling expressions up to 99.6% when labeling programs. The interpretable white-box models obtained provide information about the syntax patterns used by expert and novice programmers. By using a probabilistic classifier, it is also possible to score the expertise level of programmers regarding the syntax patterns used in their code.

Chapter 8

Future Work

This work open new future lines of research that we plan to work on. What follows is a brief description of such works.

8.1 Implementation of analyses not provided by other tools

There is plenty of documentation about common mistakes in Java code that are not detected by existing static program analysis tools. One example is the set of analyses described by Joshua Bloch for effective Java programming [63]. Some of these analyses are checked by tools, but most of them are not. When ProgQuery was designed, we considered all the items in that book, so that they could be implemented in our system. Future work is to provide such implementations.

8.2 Automatic insertion of open source code

We plan to use ProgQuery for sharing syntactic and semantic program information of existing open-source repositories over the web. ProgQuery will automatically download the source code, compile it, populate a graph database with syntactic and semantic representations, and provide that information online, as computable data. Currently, we only support GitHub projects that describe their compilation process in the Maven build automation tool. Our intention is to extend this mechanism to other repositories and build automation tools. Then, ProgQuery could transparently include in its database new open-source projects added to existing repositories.

8.3 Semantic web contents from open-source repositories

In our current implementation, we use Neo4j and Cypher because of the runtime performance they provide. However, when program representations are to be

offered as semantic web resources, other standard representations such as RDF and OWL are commonly used. Therefore, once we implement the automatic code insertion mechanism described in the previous subsection, standard semantic web representations for programs could be generated [105]. For example, the ontology defined in Appendix A could be expressed in RDFS, and all the programs in the database could be translated into RDF content.

8.4 Predictive models with semantic features

In the second case scenario presented in this dissertation, we build a Java source code classifier by considering different compound syntactic constructs. Although that kind of information seems to be sufficient to tell the difference between beginners and expert programmers, other kind of models may take advantage of semantic information. Some existing works have already used semantic program information for other purposes such as deobfuscation, advanced code completion, optimization, traceability and code fixing [106]. However, these works just extract the semantic information that the researchers think will be valuable to predict the target. ProgQuery allows extracting combined syntactic and semantic information, providing numerous additional features. The generated datasets could be used to build new predictive models aimed at improving software development.

8.5 Graph structure mining and classification

Traditional machine learning algorithms require instances/individuals to be represented as n-dimensional vectors. However, program representations are commonly modeled as graphs. There exist alternative structured prediction methods such as Graph Neural Networks (GNNs) and Conditional Random Fields (CRFs), but they suffer high computation and space costs to be used with massive codebases [96]. Moreover, they do not build interpretable white-box models. Multi-relational (MR) data mining is aimed at obtaining white-box models (MR association rules, MR decision trees and MR distance-based methods) for datasets stored in multiple tables [107]. Similarly, graph-based data mining does the same for graph structures [108]. We plan to work on adapting the existing research works to achieve mining and classifying massive graph databases, and use them with the program representations defined in this dissertation.

8.6 New programming languages and representations

ProgQuery could be extended to support other languages (e.g., Python, C#, C++ and JavaScript). For compiled languages (C++ and C), we could follow the same approach as Java, extending an existing compiler. For other languages with more advanced reflection features (Python and JavaScript), ASTs could be obtained using meta-programming. The ontology should be extended to represent particular syntax elements of each language. Although semantic representations

were designed to model elements of most imperative object-oriented languages, it may be necessary to adapt them.

The designed ontology can also be extended with new semantic representations. First, points-to information could be added. The points-to analysis provides information about the memory address a pointer expression may be pointing to [109]. Then, we could include pointer and escape information, as an extension of the points-to analysis, to offer information about which objects may escape from one program region [110]. Relying on these new representations, purity and side-effect information would provide write-effect data about methods and parameters [111]. All the representations mentioned provide valuable information for various analyses regarding object mutability, actual methods invoked (considering dynamic types of objects), object lifecycle and object aliasing. These representations would allow the user to create new analyses and improve the completeness of the existing ones (e.g., OBJ-50, OBJ-56, ERR-54 and MET-52 [18]).

8.7 Automatic error fixing

In the first case scenario presented in this dissertation, we implemented in ProgQuery 13 analyses taken from [18]. These analyses are Cypher queries that show warning messages for potential programming errors, not detected by compilers. Those messages locate the error in the input files, and give information to help programmers to fix them manually. The next step is to enhance those analyses by specifying graph transformations to automatically fix the errors. New analyses would transform ASTs it into other ASTs without errors, in just the same way existing IDEs suggest fixes for some compiler errors.

In addition to the mentioned deterministic approach to fix programs, a machine learning approach could be used [112]. Different program versions could be obtained by including plug-ins in IDEs. With this approach, we can see how syntax and semantic constructs commonly evolve from erroneous code to a final version fixed by the programmer. One example of this approach is the Nate tool, that learns how to associate ASTs of erroneous expressions to fixed ones [49]. We could extend this approach to different language constructs (no just expressions), and improve its accuracy by using semantic information (not just syntactic one).

8.8 Project information and evolution

Currently, ProgQuery represents programs as isolated graphs. However, existing repositories provide additional information about projects (e.g., owner, language, documentation, contributors and network graphs), including how the code evolves (e.g., forks, branches, pull requests and releases). We could include in our ontology new elements to model all this information. For example, programs starting with a fork from another project could be modeled with specific edges between projects. This new information would be useful for advanced code queries based on program evolution, authorship, documentation, etc. [113].

From the big code perspective, this information could be used to create pre-

dictive models, correlate program information, mine code evolution patterns and predict changes in code. For example, source code is commonly changed to support new requirements and fix errors. By analyzing how projects have evolved, new code patterns to make code more maintainable could be discovered. We could also analyze whether there exists correlation between some project properties (the use of testing code, existing documentation, the number of developers and how repositories are used) and, for instance, the number of bugs detected [114].

Appendix A

Graph Representations used in the Design of ProgQuery

In this appendix, we describe the ontology defined to represent syntactic and semantic information of Java programs. Next section describes the nodes (concepts) used in the seven representations defined in ProgQuery. Then, we detail the concepts, relationships and properties (attributes) of each representation. For more information about ProgQuery, please check [115].

A.1 Nodes

Figure A.1 shows the nodes used for the seven graph representations described in [116]. 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).

A.2 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 A.1), 100 relationships and 26 properties. They define common syntax elements of an object-oriented language [67].

A.2.1 Nodes

Leaf nodes in Figure A.1 represent concrete nodes of the AST. When a program is represented, all the particular nodes in the AST are instances of these concrete

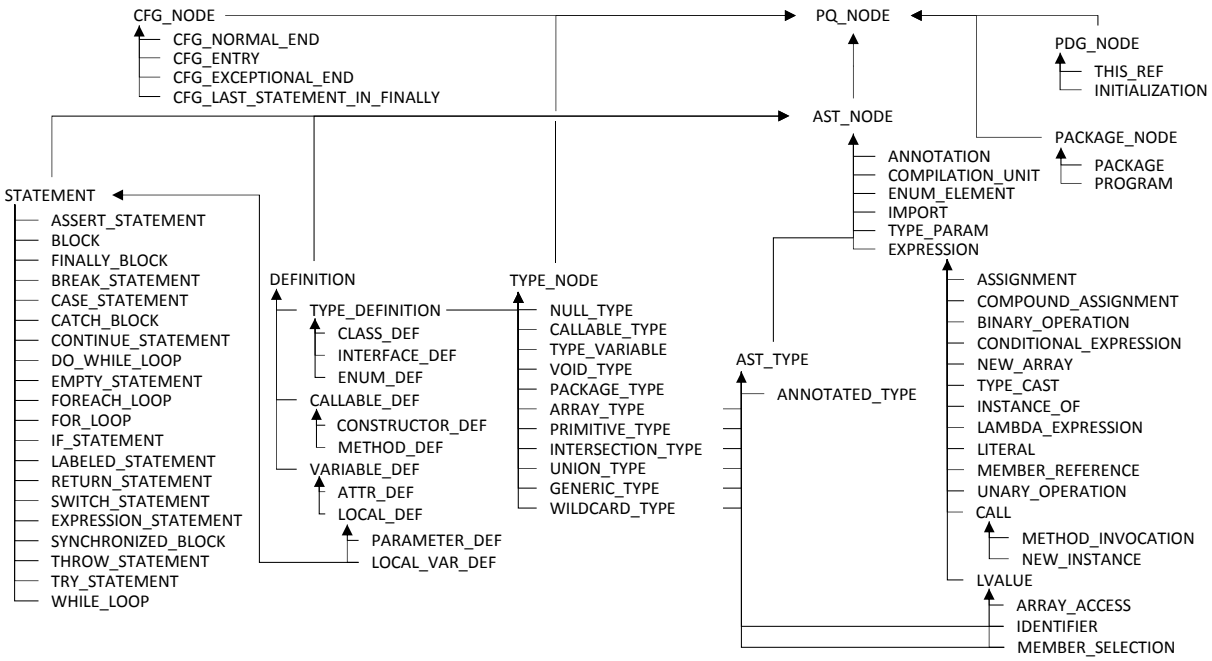


Figure A.1: Labels defined to categorize the nodes used for the different Java program representations.

labels. The rest of labels in Figure A.1 are used to generalize/classify nodes. These are the labels defined to represent ASTs:

- `ANNOTATION`: concrete node type that represents any Java annotation.
- `COMPILATION_UNIT`: Java files, which are the root nodes in ASTs (see [116]).
- `ENUM_ELEMENT`: elements included in an `enum` definition.
- `IMPORT`: import clauses used in the Java source code.
- `TYPE_PARAM`: type parameters used when a generic type, method or constructor is defined.
- `EXPRESSION`: this label is a generalization of all entities representing expressions in the AST. These are the expressions defined:
  - `ASSIGNMENT`: non-compound assignment expressions.
  - `COMPOUND_ASSIGNMENT`: Java compound assignment expressions (e.g., `+=`, `*=`, `&=` and `>>=`).
  - `BINARY_OPERATION`: binary arithmetic, logical, bitwise and relational expressions.
  - `CONDITIONAL_EXPRESSION`: ternary conditional expression($expr_1$ `?` $expr_2$ `:` $expr_3$).
  - `NEW_ARRAY`: array creation expression.
  - `TYPE_CAST`: cast expression (explicit type conversion).
  - `INSTANCE_OF`: expressions created with the `instanceof` operator.

- `LAMBDA_EXPRESSION`: represents lambda expressions, including its parameters and body.
- `LITERAL`: Java literals for built-in types, `String` and `null`.
- `MEMBER_REFERENCE`: method reference expression created with the `::` operator.
- `UNARY_OPERATION`: unary arithmetic, logical, increment, decrement and bitwise expressions.
- `CALL`: generalization of method invocation and object creation expressions:
  - `METHOD_INVOCATION`: method invocation expressions.
  - `NEW_INSTANCE`: object creation by calling the constructor via `new`.
- `LVALUE`: generalization of lvalue expressions; i.e., those Java expressions that could be placed as left-hand side of assignments:
  - `ARRAY_ACCESS`: array indexing expression, used to get one element collected by an array through the `[ ]` operator.
  - `IDENTIFIER`: variable, method and type expressions; this node also has the `AST_TYPE` label (see Figure A.1).
  - `MEMBER_SELECTION`: represents expressions created with the `.` operator: field (`obj.field`) access, method selection in method invocation (`obj.m` in `obj.m()`), full name qualifiers (`java.util.List`) and nested type access (`new OuterClass.InnerClass()`). This node also has the `AST_TYPE` label (see Figure A.1).

- `DEFINITION`: generalization of all the elements that can be defined, i.e. types (classes, interfaces and enumerations), methods, constructors and variables (attributes, parameters and local variables):
  - `TYPE_DEFINITION`: generalization to group class, enumeration and interface definitions:
    - `CLASS_DEF`: class definition.
    - `INTERFACE_DEF`: interface definition.
    - `ENUM_DEF`: definition of an enumeration.
  - `CALLABLE_DEF`: generalization of method and constructor definitions:
    - `CONSTRUCTOR_DEF`: constructor definition.
    - `METHOD_DEF`: method definition.
  - `VARIABLE_DEF`: generalization of field, parameter and local variable definitions:
    - `ATTR_DEF`: attribute (field) definition.
    - `LOCAL_DEF`: generalization of variables defined in a local scope:

        - PARAMETER_DEF: definition of a function formal parameter.
        - LOCAL_VAR_DEF: local variable definition; this node also has the STATEMENT label (see Figure A.1).
- AST_TYPE: generalization of types that could be written in the source code (and hence belong to the AST):
    - ANNOTATED_TYPE: type that has been added one or more annotations.
    - ARRAY_TYPE: represents an array type .
    - PRIMITIVE_TYPE: primitive/built-in Java type.
    - INTERSECTION_TYPE: intersection type created with the & type constructor.
    - UNION_TYPE: union type created with the | type constructor, used to catch exceptions of different types.
    - GENERIC_TYPE: an instantiated generic type; i.e, Type<*typelist*>.
    - WILDCARD_TYPE: Java wildcard type created with ? as a special type parameter.
- STATEMENT
    - ASSERT_STATEMENT: Java assert statements.
    - BLOCK: a block is a sequence of statements between { and }.
    - FINALLY_BLOCK: the `finally` clause, including the statements in the block.
    - BREAK_STATEMENT: Java `break` statement, which might include a label.
    - CASE_STATEMENT: `case` labels in a `switch` statement.
    - CATCH_BLOCK: the `catch` clause, including the statements in the block.
    - CONTINUE_STATEMENT: `continue` statement, which might include a label.
    - DO_WHILE_LOOP: includes the condition and the block.
    - EMPTY_STATEMENT: when the programmer writes a single ; as a statement.
    - FOREACH_LOOP: enhanced `for` loop with for-each semantics.
    - FOR_LOOP: classical `for` loop.
    - IF_STATEMENT: includes the condition and the `if` and `else` blocks.
    - LABELED_STATEMENT: Java labeled statements.
    - RETURN_STATEMENT: has an optional expression to be returned.
    - SWITCH_STATEMENT: it holds the condition expression and a sequence of `case` statements.

- `EXPRESSION_STATEMENT`: an expression converted into a statement; they may be simple or compound assignments, unary increments and decrements, and calls.
- `SYNCHRONIZED_BLOCK`: it holds the expression representing the monitor and the code/block with mutual exclusion.
- `THROW_STATEMENT`: encloses the expression to be thrown.
- `TRY_STATEMENT`: collects the `try`, `catch` and `finally` blocks.
- `WHILE_LOOP`: holds the condition expression and the loop body.

A.2.2 Relationships

These are the relationships defined for the AST (their domain, range and cardinality are defined in Tables A.1 and A.2):

- `ARRAYACCESS_EXPR`: relates an array access to its first child, an expression which type is array.
- `ARRAYACCESS_INDEX`: relates an array access to its index expression.
- `ASSIGNMENT_LHS`: relates an assignment to its left-hand side.
- `ASSIGNMENT_RHS`: relates an assignment to its right-hand side.
- `BINOP_LHS`: relates a binary operation to its left-hand side.
- `BINOP_RHS`: relates a (non logical) binary operation to its right-hand side.
- `BINOP_COND_RHS`: relates a logical binary operation to its right-hand side (which may be not computed).
- `CAST_ENCLOSES`: relates a type cast to the enclosed expression.
- `CAST_TYPE`: relates a type cast to the type of the coerced expression.
- `COMPOUND_ASSIGNMENT_LHS`: relates a compound assignment to its left-hand side.
- `COMPOUND_ASSIGNMENT_RHS`: relates a compound assignment to its right-hand side.
- `CONDITIONAL_EXPR_CONDITION`: relates a conditional expression (ternary operator) to its condition.
- `CONDITIONAL_EXPR_THEN`: relates a conditional (ternary) expression to the expression evaluated if the condition holds.
- `CONDITIONAL_EXPR_ELSE`: relates a conditional (ternary) expression to the expression evaluated if the condition does not hold.
- `INSTANCE_OF_EXPRESSION`: relates an `instanceof` expression to its child expression.

- `INSTANCE_OF_TYPE`: relates an `instanceof` expression to its child node representing the type.
- `LAMBDA_EXPRESSION_BODY`: relates a lambda expression to its body.
- `LAMBDA_EXPRESSION_PARAMETERS`: relates a lambda expression to its parameters, if any.
- `MEMBER_REFERENCE_EXPRESSION`: relates a member reference to the first operand (expression).
- `MEMBER_REFERENCE_TYPE_ARGUMENTS`: relates a member reference to its type arguments, if any.
- `MEMBER_SELECT_EXPR`: relates a member selection to the first operand (expression).
- `METHODINVOCATION_ARGUMENTS`: relates a method invocation to its arguments, if any.
- `METHODINVOCATION_METHOD_SELECT`: in `obj.method(args)`, relates such method invocation to `obj.method`.
- `METHODINVOCATION_TYPE_ARGUMENTS`: relates a method invocation to its type arguments, if any.
- `NEW_CLASS_ARGUMENTS`: relates a `new` instance expression to its arguments, if any.
- `NEW_CLASS_BODY`: relates a `new` instance expression to the class body defined when an anonymous class is being instantiated, if so.
- `NEW_CLASS_TYPE_ARGUMENTS`: relates a `new` instance expression to its type arguments, if any.
- `NEW_ARRAY_DIMENSION`: relates a `new` array expression to its declared dimensions, if any.
- `NEW_ARRAY_INIT`: relates a `new` array expression to its initializer expressions (i.e., between { and }) when an explicit initialization is included.
- `NEW_ARRAY_TYPE`: relates a `new` array expression to its declared type.
- `NEWCLASS_ENCLOSING_EXPRESSION`: relates a `new` class expression to its enclosing expression (i.e., for nested inner classes, `expression` is the enclosing expression of `expression.new Class(args)`).
- `UNARY_ENCLOSES`: relates a unary operation to its child expression.
- `NEWCLASS_IDENTIFIER`: relates a `new` class expression to its class identifier referencing the type to be instantiated.
- `ASSERT_CONDITION`: relates an assert statement to its condition.
- `ASSERT_DETAIL`: relates an assert statement to its message.
- `CATCH_ENCLOSES_BLOCK`: relates a `catch` statement to its block.

- `CATCH_PARAM`: relates a `catch` statement to its parameter.
- `WHILE_CONDITION`: relates a `while` statement to its condition.
- `DO_WHILE_CONDITION`: relates a `do-while` statement to its condition.
- `FOREACH_EXPR`: relates a for-each statement to its iteration expression.
- `FOREACH_STATEMENT`: relates a for-each statement to its enclosed statement or block.
- `FOREACH_VAR`: relates a for-each statement to its iteration variable.
- `FORLOOP_CONDITION`: relates a for statement to its condition.
- `FORLOOP_INIT`: relates a for statement to its initialization statements, if any.
- `FORLOOP_STATEMENT`: relates a for statement to its enclosed statement or block.
- `FORLOOP_UPDATE`: relates a for statement to its update statements, if any.
- `CASE_EXPR`: relates a `case` statement to its expression.
- `CASE_STATEMENTS`: relates a `case` statement to its statements, if any.
- `IF_CONDITION`: relates an `if` statement to its condition.
- `IF_ELSE`: relates an `if` statement to its `else` part, if any.
- `IF_THEN`: relates an `if` statement to its *then* part.
- `SWITCH_ENCLOSES_CASE`: relates a `switch` statement to its `case`s, if any.
- `SWITCH_EXPR`: relates a `switch` statement to its comparison expression.
- `SYNCHRONIZED_ENCLOSES_BLOCK`: relates a synchronized statement to its enclosed block.
- `SYNCHRONIZED_EXPR`: relates a synchronized statement to its expression.
- `THROW_EXPR`: relates a `throw` statement to the expression to be thrown.
- `TRY_BLOCK`: relates a `try` statement to its `try` block.
- `TRY_CATCH`: relates a `try` statement to its `catch` statements, if any.
- `TRY_FINALLY`: relates a `try` statement to its `finally` block, if any.
- `TRY_RESOURCES`: relates a `try` statement to its `java.lang.AutoCloseable` resources, if any.
- `LABELED_STMT_ENCLOSES`: relates a labeled statement to its statement.
- `RETURN_EXPR`: relates a `return` statement to the returned expression.
- `ENCLOSES`: relates a block to the statements it contains, if any.
- `ENCLOSES_EXPR`: when an expression is represented as a statement, this relationship connects the statement to the expression.

- `WHILE_STATEMENT`: relates a `while` loop to the enclosed statement or block.
- `DO_WHILE_STATEMENT`: relates a `do-while` loop to the enclosed statement or block.
- `IMPORTS`: relates a compilation unit to its imports, if any.
- `HAS_TYPE_DEF`: relates a compilation unit to each type definition included, if any.
- `HAS_ANNOTATIONS`: relates a definition (type, callable or variable) or annotated type to its annotations, if any.
- `HAS_ANNOTATIONS_ARGUMENTS`: relates an annotation to its arguments, if any.
- `HAS_ANNOTATION_TYPE`: relates an annotation to its annotation type.
- `HAS_EXTENDS_CLAUSE`: relates a class or interface definition to the extended types (in the `extends` clause).
- `HAS_IMPLEMENTS_CLAUSE`: relates a class or enum definition to its `implements` clauses, if any.
- `HAS_CLASS_TYPEPARAMETERS`: relates a type definition to its declared type parameters, if any.
- `DECLARES_FIELD`: relates a type definition to its declared fields, if any.
- `DECLARES_METHOD`: relates a type definition to its declared methods, if any.
- `DECLARES_CONSTRUCTOR`: relates a class or enum definition to its declared constructors, if any.
- `HAS_ENUM_ELEMENT`: relates an enum definition to its declared elements, if any.
- `UNDERLYING_TYPE`: relates an annotated type to the underlying type being annotated.
- `HAS_DEFAULT_VALUE`: relates a method definition to its default value, if any.
- `CALLABLE_HAS_BODY`: relates a callable definition to its declared body, if any.
- `CALLABLE_HAS_PARAMETER`: relates a callable definition to its declared parameters, if any.
- `CALLABLE_RETURN_TYPE`: relates a callable definition to its declared return type.
- `CALLABLE_HAS_THROWS`: relates a callable definition to its `throws` clauses.
- `CALLABLE_HAS_TYPEPARAMETERS`: relates a callable definition to its declared type parameters, if any.
- `HAS_RECEIVER_PARAMETER`: relates a callable definition to its receiver parameter, if any.
- `HAS_STATIC_INIT`: relates a class or enum definition to its static initializer, if any.

- `HAS_VARIABLEDECL_INIT`: relates a variable definition to its initialization, if any.
- `HAS_VARIABLEDECL_TYPE`: relates a variable definition to its declared type.
- `INITIALIZATION_EXPR`: relates an initialization to the initializer expression.
- `INTERSECTION_COMPOSED_OF`: relates an intersection type to the types comprising the intersection type.
- `PARAMETERIZED_TYPE`: relates a generic type to the type to parameterize.
- `GENERIC_TYPE_ARGUMENT`: relates a generic type to its type arguments.
- `TYPEPARAMETER_EXTENDS`: relates a type parameter to its `extends` bounds, if any.
- `UNION_TYPE_ALTERNATIVE`: relates a union type to its types comprising the union type.
- `WILCARD_BOUND`: relates a wildcard type to its type bound.

ProgQuery also implements the following user-defined functions:

- `database.procedures.getEnclosingClass`: relates a statement or variable definition to its enclosing class. Domain: `STATEMENT` ∪ `VARIABLE_DEF`, range: `TYPE_DEFINITION`, cardinality: 1.
- `database.procedures.getEnclosingMethod`: relates a statement or parameter to the method or constructor in which they are enclosed. Domain: `STATEMENT` ∪ `PARAMETER_DEF`, range: `CALLABLE_DEF`, cardinality: 1.
- `database.procedures.getEnclMethodFromExpr`: relates expressions to the method or constructor containing the statement in which they are enclosed. Domain:`EXPRESSION`, range: `CALLABLE_DEF`, cardinality: 0..1.
- `database.procedures.getEnclosingStmt`: relates expressions to the statement in which they are enclosed; attribute initialization expressions are related to their attribute definition. Domain: `EXPRESSION`, range: `STATEMENT` ∪ `ATTR_DEF`, cardinality: 1.

A.2.3 Properties

The following properties were defined (detailed in Table A.3):

- `lineNumber`: the line number of this node of the AST.
- `column`: column number of this node of the AST.
- `position`: position of this node in the AST nodes list.
- `isDeclared`: holds whether a specific AST element (or package) is declared in the project.
- `isAbstract`: holds if a class, interface or method is declared as `abstract`.
- `isNative`: holds if method is declared as `native`.

Relationship	Domain	Range	Cardinality
ARRAYACCESS_EXPR	ARRAY_ACCESS	EXPRESSION	1
ARRAYACCESS_INDEX	ARRAY_ACCESS	EXPRESSION	1
ASSIGNMENT_LHS	ASSIGNMENT	LVALUE	1
ASSIGNMENT_RHS	ASSIGNMENT	EXPRESSION	1
BINOP_LHS	BINARY_OPERATION	EXPRESSION	1
BINOP_RHS	BINARY_OPERATION	EXPRESSION	0..1
BINOP_COND_RHS	BINARY_OPERATION	EXPRESSION	0..1
CAST_ENCLOSES	TYPE_CAST	EXPRESSION	1
CAST_TYPE	TYPE_CAST	AST_TYPE	1
COMPOUND_ASSIGNMENT_LHS	COMPOUND_ASSIGNMENT	LVALUE	1
COMPOUND_ASSIGNMENT_RHS	COMPOUND_ASSIGNMENT	EXPRESSION	1
CONDITIONAL_EXPR_CONDITION	CONDITIONAL_EXPRESSION	EXPRESSION	1
CONDITIONAL_EXPR_THEN	CONDITIONAL_EXPRESSION	EXPRESSION	1
CONDITIONAL_EXPR_ELSE	CONDITIONAL_EXPRESSION	EXPRESSION	1
INSTANCE_OF_EXPRESSION	INSTANCE_OF	EXPRESSION	1
INSTANCE_OF_TYPE	INSTANCE_OF	AST_TYPE − PRIMITIVE_TYPE	1
LAMBDA_EXPRESSION_BODY	LAMBDA_EXPRESSION	EXPRESSION ∪ BLOCK	1
LAMBDA_EXPRESSION_PARAMETERS	LAMBDA_EXPRESSION	PARAMETER_DEF	0..*
MEMBER_REFERENCE_EXPRESSION	MEMBER_REFERENCE	EXPRESSION	1
MEMBER_REFERENCE_TYPE_ARGUMENTS	MEMBER_REFERENCE	AST_TYPE	0..*
MEMBER_SELECT_EXPR	MEMBER_SELECTION	EXPRESSION	1
METHODINVOCATION_ARGUMENTS	METHOD_INVOCATION	EXPRESSION	0..*
METHODINVOCATION_METHOD_SELECT	METHOD_INVOCATION	EXPRESSION	1
METHODINVOCATION_TYPE_ARGUMENTS	METHOD_INVOCATION	AST_TYPE	0..*
NEW_CLASS_ARGUMENTS	NEW_INSTANCE	EXPRESSION	0..*
NEW_CLASS_BODY	NEW_INSTANCE	CLASS_DEF	0..1
NEW_CLASS_TYPE_ARGUMENTS	NEW_INSTANCE	AST_TYPE	0..*
NEW_ARRAY_DIMENSION	NEW_ARRAY	EXPRESSION	0..*
NEW_ARRAY_INIT	NEW_ARRAY	EXPRESSION	0..*
NEW_ARRAY_TYPE	NEW_ARRAY	AST_TYPE	1
NEWCLASS_ENCLOSING_EXPRESSION	NEW_CLASS	EXPRESSION	0..1
NEWCLASS_IDENTIFIER	NEW_CLASS	IDENTIFIER ∪ MEMBER_SELECTION ∪ ANNOTATED_TYPE ∪ GENERIC_TYPE	1
UNARY_ENCLOSES	UNARY_OPERATION	EXPRESSION	1
ASSERT_CONDITION	ASSERT_STATEMENT	EXPRESSION	1
ASSERT_DETAIL	ASSERT_STATEMENT	EXPRESSION	0..1
CATCH_ENCLOSES_BLOCK	CATCH_BLOCK	BLOCK	1
CATCH_PARAM	CATCH_BLOCK	LOCAL_VAR_DEF	1
WHILE_CONDITION	WHILE_LOOP	EXPRESSION	1
DO_WHILE_CONDITION	DO_WHILE_LOOP	EXPRESSION	1
FOREACH_EXPR	FOREACH_LOOP	EXPRESSION	1
FOREACH_STATEMENT	FOREACH_LOOP	STATEMENT	1
FOREACH_VAR	FOREACH_LOOP	LOCAL_VAR_DEF	1
FORLOOP_CONDITION	FOR_LOOP	EXPRESSION	0..1
FORLOOP_INIT	FOR_LOOP	EXPRESSION_STATEMENT ∪ LOCAL_VAR_DEF	0..*
FORLOOP_STATEMENT	FOR_LOOP	STATEMENT	1
FORLOOP_UPDATE	FOR_LOOP	EXPRESSION_STATEMENT	0..*
CASE_EXPR	CASE_STATEMENT	LITERAL ∪ IDENTIFIER ∪ MEMBER_SELECTION ∪ BINARY_OPERATION ∪ CONDITIONAL_EXPRESSION ∪ TYPE_CAST	0..1
CASE_STATEMENTS	CASE_STATEMENT	STATEMENT	0..*
IF_CONDITION	IF_STATEMENT	EXPRESSION	1
IF_ELSE	IF_STATEMENT	STATEMENT	0..1
IF_THEN	IF_STATEMENT	STATEMENT	1

Table A.1: Relationships defined for ASTs (part 1).

Relationship	Domain	Range	Cardinality
SWITCH_ENCLOSES_CASE	SWITCH_STATEMENT	CASE_STATEMENT	0..*
SWITCH_EXPR	SWITCH_STATEMENT	EXPRESSION	1
SYNCHRONIZED_BLOCK	SYNCHRONIZED_STATEMENT	BLOCK	1
SYNCHRONIZED_EXPR	SYNCHRONIZED_STATEMENT	EXPRESSION	1
THROW_EXPR	THROW_STATEMENT	EXPRESSION	1
TRY_BLOCK	TRY_STATEMENT	BLOCK	1
TRY_CATCH	TRY_STATEMENT	CATCH_BLOCK	0..*
TRY_FINALLY	TRY_STATEMENT	FINALLY_BLOCK	0..1
TRY_RESOURCES	TRY_STATEMENT	LOCAL_VAR_DEF	0..*
LABELED_STMT_ENCLOSES	LABELED_STATEMENT	STATEMENT	1
RETURN_EXPR	RETURN_STATEMENT	EXPRESSION	0..1
ENCLOSES	BLOCK	STATEMENT	0..*
ENCLOSES_EXPR	EXPRESSION_STATEMENT	EXPRESSION	1
WHILE_STATEMENT	WHILE_LOOP	STATEMENT	1
DO_WHILE_STATEMENT	DO_WHILE_LOOP	STATEMENT	1
IMPORTS	COMPILATION_UNIT	IMPORT	0..*
HAS_TYPE_DEF	COMPILATION_UNIT	TYPE_DEFINITION	0..*
HAS_ANNOTATIONS	DEFINITION ∪ TYPE_PARAM ∪ ANNOTATED_TYPE	ANNOTATION	0..*
HAS_ANNOTATIONS_ARGUMENTS	ANNOTATION	LITERAL ∪ IDENTIFIER ∪ MEMBER_SELECTION ∪ BINARY_OPERATION ∪ CONDITIONAL_EXPRESSION ∪ TYPE_CAST	0..*
HAS_ANNOTATION_TYPE	ANNOTATION	IDENTIFIER ∪ MEMBER_SELECTION	1
HAS_EXTENDS_CLAUSE	CLASS_DEF ∪ INTERFACE_DEF	IDENTIFIER ∪ MEMBER_SELECTION ∪ GENERIC_TYPE	0..*
HAS_IMPLEMENTS_CLAUSE	CLASS_DEF ∪ ENUM_DEF	IDENTIFIER ∪ MEMBER_SELECTION ∪ GENERIC_TYPE	0..*
HAS_CLASS_TYPEPARAMETERS	TYPE_DEFINITION	AST_TYPE	0..*
DECLARES_FIELD	TYPE_DEFINITION	ATTR_DEF	0..*
DECLARES_METHOD	TYPE_DEFINITION	METHOD_DEF	0..*
DECLARES_CONSTRUCTOR	CLASS_DEF ∪ ENUM_DEF	CONSTRUCTOR_DEF	0..*
HAS_ENUM_ELEMENT	ENUM_DEF	ENUM_ELEMENT	0..*
UNDERLYING_TYPE	ANNOTATED_TYPE	AST_TYPE	1
HAS_DEFAULT_VALUE	METHOD_DEF	LITERAL ∪ IDENTIFIER ∪ MEMBER_SELECTION ∪ BINARY_OPERATION ∪ CONDITIONAL_EXPRESSION ∪ TYPE_CAST	0..1
CALLABLE_HAS_BODY	CALLABLE_DEF	BLOCK	0..1
CALLABLE_HAS_PARAMETER	CALLABLE_DEF	PARAMETER_DEF	0..*
CALLABLE_RETURN_TYPE	CALLABLE_DEF	AST_TYPE	1
CALLABLE_HAS_THROWS	CALLABLE_DEF	IDENTIFIER ∪ MEMBER_SELECTION	0..*
CALLABLE_HAS_TYPEPARAMETERS	CALLABLE_DEF	AST_TYPE	0..*
HAS_RECEIVER_PARAMETER	CALLABLE_DEF	PARAMETER_DEF	0..1
HAS_STATIC_INIT	CLASS_DEF ∪ ENUM_DEF	BLOCK	0..1
HAS_VARIABLEDECL_INIT	VARIABLE_DEF	INITIALIZATION	0..1
HAS_VARIABLEDECL_TYPE	VARIABLE_DEF	AST_TYPE	1
INITIALIZATION_EXPR	INITIALIZATION	EXPRESSION	1
INTERSECTION_COMPOSED_OF	INTERSECTION_TYPE	AST_TYPE	2..*
PARAMETERIZED_TYPE	GENERIC_TYPE	IDENTIFIER ∪ MEMBER_SELECTION ∪ ANNOTATED_TYPE	1
GENERIC_TYPE_ARGUMENT	GENERIC_TYPE	AST_TYPE − {PRIMITIVE_TYPE, INTERSECTION_TYPE, UNION_TYPE}	0..*
TYPEPARAMETER_EXTENDS	TYPE_PARAM	AST_TYPE − {PRIMITIVE_TYPE, ARRAY_TYPE, WILDCARD_TYPE, UNION_TYPE, INTERSECTION_TYPE}	0..*
UNION_TYPE_ALTERNATIVE	UNION_TYPE	IDENTIFIER ∪ MEMBER_SELECTION ∪ ANNOTATED_TYPE	2..*
WILCARD_BOUND	WILDCARD_TYPE	AST_TYPE − {PRIMITIVE_TYPE, INTERSECTION_TYPE, UNION_TYPE, WILDCARD_TYPE}	0..1

Table A.2: Relatioships defined for ASTs (part 2).

- `isStatic`: holds if an AST element is declared as `static`.
- `isFinal`: holds if an AST element is declared as `final`.
- `isStrictfp`: holds if a method is declared as `strictfp`.
- `isSynchronized`: holds if a method is declared as `synchronized`.
- `isTransient`: holds if a field is declared as `transient`.
- `isVolatile`: holds if a field is declared as `volatile`.
- `accessLevel`: represents the access level of a type, callable or attribute definition.
- `name`: string holding the name for identifier, variable and method definition, type parameter and package nodes.
- `memberName`: string holding the name of the accessed member.
- `completeName`: for a given method/constructor, a string with the format `java.lang.Object:equals`.
- `fullyQualifiedName`: for a given method/constructor, a string with the format `java.lang.Object:equals(java.lang.Object)`.
- `simpleName`: string holding the simple name of types.
- `packageName`: string holding the package name of each compilation unit.
- `fileName`: string holding the path and file name of each compilation unit.
- `qualifiedIdentifier`: string representing the package or class to be imported.
- `typetag`: string representing the type of literal.
- `label`: string holding the name of the label associated to a `break` or `continue` statement.
- `operator`: operator of common expressions, represented as a string.
- `argumentIndex`: integer value representing the index of an argument among all the arguments in the given method.
- `paramIndex`: Integer value representing the index of a parameter among all the parameters in the given method.

A.3 Control Flow Graph

A.3.1 Nodes

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.

Property	Type	Domain	Value-Type	Cardinality
lineNumber	Node	AST_NODE	Integer[1, Inf)	1
column	Node	AST_NODE	Integer[1, Inf)	1
position	Node	AST_NODE	Integer[1, Inf)	1
isDeclared	Node	PACKAGE ∪ TYPE_DEFINITION ∪ CALLABLE_DEF ∪ ATTR_DEF	Boolean	1
isAbstract	Node	CLASS_DEF ∪ INTERFACE_DEF ∪ METHOD_DEF	Boolean	1
isNative	Node	METHOD_DEF	Boolean	1
isStatic	Node	METHOD_DEF ∪ TYPE_DEFINITION ∪ BLOCK ∪ IMPORT ∪ ATTR_DEF	Boolean	1
isFinal	Node	METHOD_DEF ∪ TYPE_DEFINITION ∪ VARIABLE_DEF	Boolean	1
isStrictfp	Node	METHOD_DEF	Boolean	1
isSynchronized	Node	METHOD_DEF	Boolean	1
isTransient	Node	ATTR_DEF	Boolean	1
isVolatile	Node	ATTR_DEF	Boolean	1
accessLevel	Node	TYPE_DEFINITION ∪ CALLABLE_DEF ∪ ATTR_DEF	{public, protected, package, private}	1
name	Node	CALLABLE_DEF ∪ IDENTIFIER ∪ TYPE_PARAM ∪ VARIABLE_DEF ∪ LABELED_STATEMENT ∪ MEMBER_REFERENCE ∪ PACKAGE	String	1
memberName	Node	MEMBER_SELECTION	String	1
completeName	Node	CALLABLE_DEF	String	1
fullyQualifiedName	Node	TYPE_DEFINITION ∪ ARRAY_TYPE ∪ CALLABLE_TYPE ∪ PRIMITIVE_TYPE ∪ UNION_TYPE ∪ CALLABLE_DEF	String	1
simpleName	Node	TYPE_DEFINITION ∪ ARRAY_TYPE ∪ CALLABLE_TYPE ∪ PRIMITIVE_TYPE ∪ UNION_TYPE	String	1
packageName	Node	COMPILATION_UNIT	String	1
fileName	Node	COMPILATION_UNIT	String	1
qualifiedIdentifier	Node	IMPORT	String	1
typetag	Node	LITERAL	{INT_LITERAL, FLOAT_LITERAL, STRING_LITERAL, NULL_LITERAL, CHAR_LITERAL, DOUBLE_LITERAL, LONG_LITERAL}	1
label	Node	BREAK_STATEMENT ∪ CONTINUE_STATEMENT	String	1
operator	Node	BINARY_OPERATION ∪ UNARY_OPERATION ∪ COMPOUND_ASSIGNMENT	{PLUS, MINUS, DIVIDE, EQUAL_TO, PREFIX_INCREMENT...}	1
argumentIndex	Node	METHODINVOCATION_ARGUMENTS ∪ METHODINVOCATION_TYPE_ARGUMENTS ∪ NEW_CLASS_ARGUMENTS ∪ NEW_CLASS_TYPE_ARGUMENTS ∪ MEMBER_REFERENCE_TYPE_ARGUMENTS ∪ HAS_ANNOTATIONS_ARGUMENTS ∪ GENERIC_TYPE_ARGUMENTS	Integer[0, Inf)	1
paramIndex	Node	HAS_METHODDECL_PARAMETERS ∪ LAMBDA_EXPRESSION_PARAMETERS ∪ HAS_CLASS_TYPEPARAMETERS ∪ HAS_METHODDECL_PARAMETERS ∪ HAS_METHODDECL_TYPEPARAMETERS	Integer[0, Inf)	1

Table A.3: Properties defined for ASTs.

- `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.

A.3.2 Relationships

We now describe the relationships of CFG. Table A.4 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 the 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 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 if 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 `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 local variable (between `(` and `)`) if, considering the hierarchical type information, the exception could be caught.
- `CFG_MAY_THROW`: relates a 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.

ProgQuery provides the following user-defined functions:

- `database.procedures.getAnySucc`: relates a statement or control flow node to its successors, including itself. Domain: `CFG_NODE` ∪ `STATEMENT`, range: `CFG_NODE` ∪ `STATEMENT`, cardinality: 1..*.
- `database.procedures.getAnySuccNotItself`: relates a statement or control flow node to its possible successors, not including itself. Domain: `CFG_NODE` ∪ `STATEMENT`, range: `CFG_NODE` ∪ `STATEMENT`, cardinality: 0..*.

A.3.3 Properties

These are the properties of the CFG nodes and relationships (see details in Table A.5):

- `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.

A.4 Call Graph

A.4.1 Nodes

No new nodes are defined for the Call Graph.

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 A.4: Relationships 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 A.5: Properties defined for CFGs.

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..*

Table A.6: Relationships defined for Call Graphs.

A.4.2 Relationships

These are the Call Graph relationships (detailed in Table A.6):

- `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.

A.4.3 Properties

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 other initializers.

A.5 Type Graph

A.5.1 Nodes

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 a package reference expression (i.e. `java.lang`).
- `NULL_TYPE`: node representing the type of `null`.
- `PRIMITIVE_TYPE`: representation of any Java primitive type.

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 A.7: Relationships defined for Type Graphs.

- TYPE_VARIABLE: type variables used with generic types and methods.
- 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., `?`).

A.5.2 Relationships

The following relationships are defined for Type Graphs (Table A.7):

- IS_SUBTYPE_EXTENDS: relates a type definition to its direct supertypes.
- IS_SUBTYPE_IMPLEMENTS: relates a class or enum definition to its direct superinterfaces.
- 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.

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 A.8: Properties defined for Type Graphs.

- `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*).

A.5.3 Properties

The following properties are defined (Table A.8):

- `actualType`: string representing the type of an expression, callable or variable definition.
- `typeKind`: string representing a type generalization (Table A.8).
- `typeBoundKind`: string describing the kind of bound of a wildcard type (Table A.8).

A.6 Program Dependency Graph

A.6.1 Nodes

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.

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

Table A.9: Relationships defined for PDGs.

A.6.2 Relationships

Relationships defined for PDGs (Table A.9):

- `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 a type definition to the implicit object reference (`this`).

A.6.3 Properties

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`).

A.7 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..*.

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..*

Table A.10: Relationships defined for Package Graphs.

A.8 Package Graph

A.8.1 Nodes

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.

A.8.2 Relationships

What follows are the Package Graph relationships defined (details in Table A.10):

- 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.

A.8.3 Properties

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.

Appendix B

Analyses

What follows it the Cypher source code for implementing in ProgQuery the 13 analyses described in Section 5.1:

B.1 MET53-J

```
MATCH (enclosingCU)-[:HAS_TYPE_DEF | :HAS_INNER_TYPE_DEF]
    ->(typeDec)-[:DECLARES_METHOD | :DECLARES_CONSTRUCTOR]
    ->(md)-[:CALLABLE_RETURN_TYPE]->(typeRet)
WHERE NOT typeRet:PRIMITIVE_TYPE  AND
    md.fullyQualifiedName CONTAINS ':clone()'
OPTIONAL MATCH (md)-[:CALLS]->()-[:HAS_DEF]->(superDec)
    -[:CALLABLE_RETURN_TYPE]->(superRet)
WHERE  superDec.fullyQualifiedName CONTAINS ':clone()' AND
    NOT superDec.fullyQualifiedName CONTAINS
        (typeDec.fullyQualifiedName+':') AND
    NOT superRet:PRIMITIVE_TYPE
WITH enclosingCU, md, COUNT(superDec) as superCallsCount
    WHERE superCallsCount=0
RETURN 'Warning [CMU-MET53], you must call super.clone in
    every overridden clone method. Line ' + md.lineNumber +
    ' in ' + enclosingCU.fileName + '.'
```

B.2 MET55-J

```
MATCH (md)-[:CALLABLE_RETURN_TYPE]->(rt)
    -[:ITS_TYPE_IS |:PARAMETERIZEDTYPE_TYPE*0..]->()
    -[:IS_SUBTYPE_EXTENDS | :IS_SUBTYPE_IMPLEMENTS*0..]
    ->(collection)
WHERE collection.fullyQualifiedName='java.util.Collection<E>'
    OR collection:ARRAY_TYPE  WITH DISTINCT md
MATCH (enclosingCU)-[:HAS_TYPE_DEF | :HAS_INNER_TYPE_DEF]
    ->()-[:DECLARES_METHOD]->(md)<-[:CFG_END_OF]-(normalEnd)
    <-[:CFG_NEXT_STATEMENT]-(:RETURN_STATEMENT)-[:RETURN_EXPR]->()
    -[:CONDITIONAL_EXPR_THEN |:CONDITIONAL_EXPR_ELSE*0..]
```

```
    ->(nullRet{typetag:'NULL_LITERAL'})
WITH enclosingCU, nullRet  WHERE nullRet IS NOT NULL
RETURN 'Warning [CMU-MET55], you must not return null when you can
    return an empty collection or array.Line' + nullRet.lineNumber +
    ' in ' + enclosingCU.fileName + '.'
```

B.3 SEC56-J

```
MATCH (class)-[:DECLARES_FIELD]->(f{isTransient:false})-[:ITS_TYPE_IS]
    ->(aux)-[:IS_SUBTYPE_EXTENDS | :IS_SUBTYPE_IMPLEMENTS*0..]
    ->(fTypeOrSupertype),
    (class)-[:IS_SUBTYPE_EXTENDS | :IS_SUBTYPE_IMPLEMENTS*]
    ->(superInt:INTERFACE_DEF{fullyQualifiedName:'java.io.Serializable'})
WHERE fTypeOrSupertype.fullyQualifiedName IN ['java.io.File',
    'org.omg.CosNaming.NamingContext' ,'org.omg.CORBA.DomainManager',
    'org.omg.PortableInterceptor.ObjectReferenceFactory']
RETURN DISTINCT 'Warning [CMU-SEC56], you must not serialize direct handles
    to system resources like field ' + f.name + '(an instance of ' +
    fTypeOrSupertype.fullyQualifiedName + '). Line ' + f.lineNumber +
    ' in ' + class.fullyQualifiedName + '.'
```

B.4 DCL56-J

```
MATCH (enclosingCU)-[:HAS_TYPE_DEF | :HAS_INNER_TYPE_DEF]->()
    -[:DECLARES_METHOD | :DECLARES_CONSTRUCTOR]->(enclosingM)
    -[:CALLS]->(inv)-[:HAS_DEF]->(md)
WHERE md.fullyQualifiedName ='java.lang.Enum:ordinal()int'
RETURN 'Warning [CMU-DEC56], you should not attach significance to the
    ordinal of an enum. Line ' + inv.lineNumber + ' in ' +
    enclosingCU.fileName + '.'
```

B.5 MET50-J

```
MATCH (enclosingCU)-[:HAS_TYPE_DEF | :HAS_INNER_TYPE_DEF]->(class)
    -[:DECLARES_METHOD | :DECLARES_CONSTRUCTOR]->(md)
    -[:CALLABLE_HAS_PARAMETER]->(p:PARAMETER_DEC)
WITH enclosingCU, md, class, COLLECT(p.actualType) as params
MATCH (class)-[:DECLARES_METHOD | :DECLARES_CONSTRUCTOR]->(md2)
    -[:CALLABLE_HAS_PARAMETER]->(p2)
WHERE md.name = md2.name AND md <> md2
WITH  enclosingCU, md, md2, params, COLLECT(p2.actualType) as params2
WHERE ALL(p IN params WHERE p IN params2) AND
    all(p IN params2 WHERE p IN params) AND
    ((SIZE(params)>=4 AND SIZE(params2)>=4) OR  SIZE(params)=SIZE(params2))
RETURN 'Warning [CMU-MET50], you must avoid confusing overloadings like ' +
    md.fullyQualifiedName + ' in line ' + md.lineNumber +
    ' ( very similar to declaration in line ' + md2.lineNumber + ') in ' +
    enclosingCU.fileName + '.'
```

B.6 DCL60-J

```
MATCH (package1)-[:DEPENDS_ON_PACKAGE]->(package2),p=(package2)
    -[:DEPENDS_ON_PACKAGE*]->(package1)
WITH REDUCE(warning='', package IN package1 + NODES(p)| warning +
    '->'  +package.name) as packageDepList
RETURN 'Warning [CMU-DCL60] There is a cycle between packages
    caused by the dependencies between ' +
    SUBSTRING(packageDepList,2,LENGTH(packageDepList)) +
    '. You should undo them.'
```

B.7 OBJ54-J

```
MATCH (varDec:LOCAL_DEF)-[:MODIFIED_BY]->(ass:ASSIGNMENT)
    -[:ASSIGNMENT_RHS]->(:LITERAL{typeKind:'NULL'})
OPTIONAL MATCH (varDec)-[:USED_BY|STATE_MODIFIED_BY]->(use)
WITH varDec,database.procedures.getEnclosingStmt(ass) as assStat,
    COLLECT(database.procedures.getEnclosingStmt(use)) as useStats
WHERE SIZE(FILTER( succ IN database.procedures.getAnySucc(assStat)
    WHERE succ IN useStats))=0
RETURN 'Warning [CMU-OBJ54] You must not try to help garbage collector
    setting references to null when they are no longer used.
    To make your code clearer, just delete the assignment in line ' +
    assStat.lineNumber + ' of the variable ' + varDec.name +
    ' declared in class ' + database.procedures.getEnclosingClass(varDec)
    .fullyQualifiedName + '.'
```

B.8 OBJ50-J

```
MATCH (variable:VARIABLE_DEF {isFinal:true})
    -[mutation:STATE_MODIFIED_BY|STATE_MAY_BE_MODIFIED_BY]
    ->(mutatorExpr)
WITH variable, mutation, mutatorExpr,
    database.procedures.getEnclMethodFromExpr(mutatorExpr)
    as mutatorMethod
MATCH (mutatorMethod)
    <-[:DECLARES_METHOD| DECLARES_CONSTRUCTOR|HAS_STATIC_INIT]
    -(mutatorEnclClass)<-[:HAS_TYPE_DEF|:HAS_INNER_TYPE_DEF]
    -(mutatorCU:COMPILATION_UNIT)
WHERE NOT(variable:ATTR_DEF AND mutation.isOwnAccess
    AND mutatorMethod.isInitializer)
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
MATCH (variableEnclClass)<-[:HAS_TYPE_DEF| :HAS_INNER_TYPE_DEF]
    -(variableCU:COMPILATION_UNIT)
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
```

B.9 ERR54-J

```
MATCH (closeableSubtype:TYPE_DEFINITION)
    -[:IS_SUBTYPE_EXTENDS | :IS_SUBTYPE_IMPLEMENTS*0..]
    ->(closeableInt:INTERFACE_DEF{fullyQualifiedName:
    'java.lang.AutoCloseable'})
WITH DISTINCT closeableSubtype.fullyQualifiedName as className
MATCH (closeableDec{actualType:className})-[:MODIFIED_BY]->(assign)
    WHERE  closeableDec:VAR_DEC
OPTIONAL MATCH (closeableDec)<-[r:TRY_RESOURCES]-()
WITH database.procedures.getEnclosingStmt(assign) as assignStat,
    r, closeableDec
WHERE r IS NULL UNWIND database.procedures.
    getAnySuccNotItself(assignStat) as prev
OPTIONAL MATCH (mInv:METHOD_INVOCATION)-[:METHODINVOCATION_METHOD_SELECT]
    ->(mSelect:MEMBER_SELECTION{memberName:'close'})-[:MEMBER_SELECT_EXPR]
    ->(id)<-[:USED_BY]-(closeableDec),(prev)
    -[exceptionRel:CFG_IF_THERE_IS_UNCAUGHT_EXCEPTION|
        :CFG_MAY_THROW|:CFG_THROWS]
    ->(afterEx)
WITH  COLLECT(DISTINCT database.procedures.getEnclosingStmt(mInv))
    as closes, prev,closeableDec, afterEx,exceptionRel
WHERE NOT prev IN closes AND
    ANY(prevSucc IN database.procedures.getAnySuccNotItself(prev)
        WHERE prevSucc IN closes)
OPTIONAL MATCH p=(afterEx)-[:CFG_NEXT_STATEMENT|:CFG_NEXT_STATEMENT_IF_TRUE|
    :CFG_NEXT_STATEMENT_IF_FALSE | :CFG_FOR_EACH_HAS_NEXT |
    :CFG_FOR_EACH_NO_MORE_ELEMENTS | :CFG_IF_THERE_IS_UNCAUGHT_EXCEPTION |
    :CFG_NO_EXCEPTION|:CFG_CAUGHT_EXCEPTION|:CFG_AFTER_FINALLY_PREVIOUS_BREAK|
    :CFG_AFTER_FINALLY_PREVIOUS_CONTINUE | :CFG_SWITCH_CASE_IS_EQUAL_TO |
    :CFG_SWITCH_DEFAULT_CASE | :CFG_MAY_THROW | :CFG_THROWS *0..]
    ->(reachableAfterEx)
WHERE reachableAfterEx IN closes
WITH prev as prevs, closeableDec,p,exceptionRel,
    CASE WHEN p IS NULL THEN NULL
                ELSE EXTRACT (index IN RANGE(0,SIZE(NODES(p)))
    | index=0 OR TYPE(RELATIONSHIPS(p)[index-1]) IN
    ['CFG_THROWS' ,'CFG_IF_THERE_IS_UNCAUGHT_EXCEPTION' ,'CFG_MAY_THROW' ])
    END as previousThrow
WITH prevs, closeableDec,p,exceptionRel,
    CASE WHEN p IS NULL THEN NULL
    ELSE EXTRACT (index IN RANGE(0,SIZE(NODES(p))) |
        CASE WHEN NODES(p)[index]:CATCH_BLOCK OR previousThrow[index] AND
            NODES(p)[index]:LOCAL_VAR_DEF   THEN 'catch'
        ELSE CASE WHEN previousThrow[index] THEN
```

```
            CASE WHEN index=0 THEN exceptionRel.exceptionType
            ELSE RELATIONSHIPS(p)[index-1].exceptionType
            END ELSE
        CASE WHEN NODES(p)[index]:TRY_STATEMENT THEN 'newtry'
        ELSE  NULL
        END END
    END)
    END as exFlow
WITH p,closeableDec,prevs,
    CASE WHEN p IS NULL THEN NULL
    ELSE EXTRACT( relIndex IN RANGE(0,SIZE(RELATIONSHIPS(p))) |
        exFlow[LAST(FILTER( exIndex IN RANGE(0,SIZE(exFlow))
        WHERE exIndex<=relIndex AND NOT exFlow[exIndex] IS NULL))])
    END as exFlow
WITH closeableDec,prevs, NOT ANY(x IN COLLECT(CASE WHEN p IS NULL
    THEN FALSE ELSE ALL(relIndex IN RANGE(0,SIZE(RELATIONSHIPS(p)))
    WHERE CASE WHEN TYPE(RELATIONSHIPS(p)[relIndex])='CFG_NO_EXCEPTION'
    THEN exFlow[relIndex] IN ['catch' , 'newtry']
    ELSE CASE WHEN TYPE(RELATIONSHIPS(p)[relIndex])=
    'CFG_IF_THERE_IS_UNCAUGHT_EXCEPTION'
    THEN exFlow[relIndex]=RELATIONSHIPS(p)[relIndex].exceptionType
    ELSE TRUE END END )
    END)
    WHERE x) as truePathToClose
WITH closeableDec,COLLECT(prevs) as prevs,
  COLLECT(truePathToClose) as truePathToClose
  WHERE ANY (x IN truePathToClose WHERE x)
RETURN  'Warning [CMU-ERR54] variable ' + closeableDec.name +
    '(defined in line'+closeableDec.lineNumber+', class ' +
    database.procedures.getEnclosingClass(closeableDec).
    fullyQualifiedName + ') might not be properly closed,
    as statement(s) (in lines ' +
    EXTRACT(prev IN prevs | prev.lineNumber) +
    ') may throw an exception.'
```

B.10 MET52-J

```
MATCH (enclosingCU)-[:HAS_TYPE_DEF |:HAS_INNER_TYPE_DEF]
    ->(typeDec{accessLevel:'public'})-[:DECLARES_METHOD]
    ->(method{accessLevel:'public'})-[:CALLABLE_HAS_PARAMETER]
    ->(param) -[:USED_BY]->(id)<-[:MEMBER_SELECT_EXPR]
    -(mSelect:MEMBER_SELECTION{memberName:'clone'})
    <-[:METHODINVOCATION_METHOD_SELECT]-(mInv:METHOD_INVOCATION),
    (param)-[:HAS_VARIABLEDECL_TYPE]->()-[:PARAMETERIZEDTYPE_TYPE*0..1]
    ->()-[:ITS_TYPE_IS]->(pType)
WHERE mSelect.actualType CONTAINS '()' AND NOT pType.isFinal AND
    (NOT pType.isDeclared OR pType.accessLevel='public')
RETURN 'Warning [CMU-MET52] You must not use the clone method to copy
    unstrusted parameters (like parameter ' + param.name +
    ', cloned in line ' +  mInv.lineNumber+ ' in method ' +
```

```
    method.name + ', file ' + enclosingCU.fileName + ').'
```

B.11 DCL53-J

```
MATCH (typeDec)-[:DECLARES_FIELD]->(attr:ATTR_DEF)
WHERE NOT ((attr.accessLevel='public' OR attr.accessLevel='protected' AND
    NOT typeDec.isFinal) AND typeDec.accessLevel='public') AND
    NOT( attr.isStatic AND attr.actualType='long' AND attr.isFinal AND
    attr.name='serialVersionUID')
OPTIONAL MATCH (attr)-[attrUseRel:USED_BY | :STATE_MODIFIED_BY]->(exprUse)
WITH typeDec.fullyQualifiedName as className,
    attr,database.procedures.getEnclosingStmt(exprUse) as exprUseStat, exprUse
    OPTIONAL MATCH (attr)-[attrModifRel:MODIFIED_BY]
        ->(modif)-[:ASSIGNMENT_LHS]->(lhs_expr)
WHERE database.procedures.getEnclosingMethod(database.procedures.
    getEnclosingStmt(modif))=database.procedures.
    getEnclosingMethod(exprUseStat) AND
    database.procedures.getEnclosingStmt(modif).position < exprUseStat.position
WITH attrModifRel,attrUseRel,typeDec.fullyQualifiedName as className,
    attr,outerBlock, exprUseStat, exprUse ,method, exprModStat,
    database.procedures.getEnclosingStmt(modif) as modif, lhs_expr
MATCH q=(method)-[:CFG_NEXT_STATEMENT | :CFG_NEXT_STATEMENT_IF_TRUE |
    :CFG_NEXT_STATEMENT_IF_FALSE | :CFG_FOR_EACH_HAS_NEXT |
    :CFG_FOR_EACH_NO_MORE_ELEMENTS | :CFG_IF_THERE_IS_UNCAUGHT_EXCEPTION |
    :CFG_NO_EXCEPTION | :CFG_CAUGHT_EXCEPTION |
    :CFG_AFTER_FINALLY_PREVIOUS_BREAK | :CFG_AFTER_FINALLY_PREVIOUS_CONTINUE |
    :CFG_SWITCH_CASE_IS_EQUAL_TO | :CFG_SWITCH_DEFAULT_CASE | :CFG_MAY_THROW |
    :CFG_THROWS | :CFG_ENTRIES *]->(exprUseStat)
WITH attrModifRel,attrUseRel,exprModStat IN NODES(q) as modInPath, className,
    attr,outerBlock, exprUseStat, exprUse ,method, exprModStat, modif, lhs_expr
OPTIONAL MATCH (exprUse)-[:MEMBER_SELECT_EXPR]->(memberSelectExprUse)
    <-[:USED_BY]-(varDec)
OPTIONAL MATCH p=(varDec)-[:STATE_MODIFIED_BY]->(modif)
    OPTIONAL MATCH (lhs_expr)-[:MEMBER_SELECT_EXPR]->(memberSelectExprModif)
WITH attrModifRel,attrUseRel,ALL( modInPath IN COLLECT(modInPath)
WHERE modInPath) as unconditionalAssign, method,p,className,
    attr.lineNumber as line, attr.name as attr,exprUse, modif, lhs_expr,
    memberSelectExprUse, memberSelectExprModif, exprUseStat, exprModStat
WITH attrModifRel,attrUseRel,className, attr,line, exprUse,
    unconditionalAssign,  attrUseRel.isOwnAccess AND
    attrModifRel.isOwnAccess OR
    (NOT p IS NULL AND NOT memberSelectExprUse IS NULL AND
        NOT memberSelectExprModif IS NULL AND
        memberSelectExprUse:IDENTIFIER AND
        memberSelectExprModif:IDENTIFIER) as isTheSameVar
WITH line,className, attr ,exprUse,
    ANY(x IN COLLECT(unconditionalAssign AND isTheSameVar) WHERE x)
        as prevAssign
WITH line,className, attr , ALL( x IN COLLECT(prevAssign) WHERE x)
    OR exprUse IS NULL as isSillyAttr WHERE isSillyAttr
```

```
RETURN 'Warning [CMU-DCL53] You must minimize the scope of the variables.
    You can minimize the scope of the attribute ' + attr +
    '(declared in line ' + line + ') in class ' + className +
    ' by transforming it into a local varaible (as everytime its value
    is used in a method, there is a previous unconditional assignment).'
```

B.12 OBJ56-J

```
MATCH (enclosingType)-[:DECLARES_FIELD]->(field:ATTR_DEF)-[:USED_BY]
    ->(retExpr)<-[:RETURN_EXPR]-(retStat)
WITH enclosingType,field,database.procedures.getEnclosingMethod(retStat)
    as method WHERE method.accessLevel='public'
MATCH (method)<-[:DECLARES_METHOD]-(classExposingF{accessLevel:'public',
    isAbstract:false})
WHERE NOT field.accessLevel='public' AND (NOT field.accessLevel='protected'
    OR classExposingF.isFinal)
MATCH (field)-[:ITS_TYPE_IS]->(fieldType)
    <-[:IS_SUBTYPE_EXTENDS|IS_SUBTYPE_IMPLEMENTS*0..]
    -(fieldTypeOrSubtype), accessibleMembers=(fieldTypeOrSubtype)
    -[:DECLARES_FIELD|ITS_TYPE_IS|INHERITS_FIELD*0..]->(accessibleMember)
WITH field,fieldType=fieldTypeOrSubtype as isFieldType,enclosingType,
    NODES(accessibleMembers) as accessibleMembers,accessibleMember,
    fieldTypeOrSubtype, COLLECT(DISTINCT method.fullyQualifiedName+'
    -> line '+method.lineNumber) as publicGetters,
    EXTRACT(index IN RANGE(0,SIZE( NODES(accessibleMembers))-1) |
        [CASE WHEN index=0 THEN field
            ELSE NODES(accessibleMembers)[index-1] END,
    NODES(accessibleMembers)[index]]) as accessibleMembersAndPrevs
WITH field,isFieldType,enclosingType, accessibleMember,fieldTypeOrSubtype,
    publicGetters,  accessibleMembersAndPrevs,
    LAST(accessibleMembersAndPrevs)[0] as accessibleMemberPrev
UNWIND accessibleMembersAndPrevs as accMemberAndPrev
OPTIONAL MATCH (accesibleField)-[:USED_BY]->(fieldExpr)<-[:RETURN_EXPR]
    -(returnStat),(method{accessLevel:'public'})
    <-[:DECLARES_METHOD|INHERITS_METHOD]-(accessibleType)
WHERE database.procedures.getEnclosingMethod(returnStat)=method AND
    accesibleField=accMemberAndPrev[1] AND
    accessibleType=accMemberAndPrev[0]
WITH  field,isFieldType,enclosingType, method,publicGetters,
    accMemberAndPrev, accessibleMember,accessibleMemberPrev,
    fieldTypeOrSubtype, COLLECT(method) as gettersForCurrentMember
WITH field,isFieldType,enclosingType,fieldTypeOrSubtype,publicGetters,
    accessibleMember,accessibleMemberPrev,
    ALL( isAccHere IN COLLECT( CASE WHEN accMemberAndPrev[1]:ATTR_DEF
    THEN NOT accMemberAndPrev[1].isStatic AND
        (accMemberAndPrev[1].accessLevel='public' OR
        SIZE(gettersForCurrentMember)>0)
    ELSE CASE WHEN accMemberAndPrev[1]:ARRAY_TYPE THEN TRUE
    ELSE EXISTS(accMemberAndPrev[1].accessLevel) AND
    accMemberAndPrev[1].accessLevel='public' END END)
```

```
    WHERE isAccHere) as isExternallyAcc
OPTIONAL MATCH (accessibleMember)-[:DECLARES_METHOD|INHERITS_METHOD]
    ->(mutator:METHOD_DEF{accessLevel:'public'})
    <-[:STATE_MAY_BE_MODIFIED_BY|STATE_MODIFIED_BY]-(:THIS_REF)
WITH field,isFieldType,fieldTypeOrSubtype,enclosingType, publicGetters,
    COLLECT(DISTINCT 'MUTATOR METHOD '+mutator.fullyQualifiedName +
    ' -> line ' + mutator.lineNumbe) as allMut,
    EXTRACT(accField IN FILTER(accMember IN COLLECT(DISTINCT accessibleMember)
    WHERE accMember:ATTR_DEF AND NOT accMember.isStatic AND
    NOT accMember.isFinal AND accMember.accessLevel='public') |
    'PUBLIC NON-FINAL FIELD ' + accField.name + '-> line' +
    accField.lineNumber) as externallyMutableFields,
    FILTER(x IN COLLECT(DISTINCT [accessibleMember:ARRAY_TYPE AND
        isExternallyAcc,'ACCESIBLE ARRAY FIELD ' + accessibleMemberPrev.name +
        '-> line ' +accessibleMemberPrev.lineNumber])
    WHERE x[0]) as arrayFieldsExposed
WITH field,isFieldType,enclosingType,publicGetters,fieldTypeOrSubtype.
    fullyQualifiedName as fieldTypeOrSubtype,allMut,externallyMutableFields,
    arrayFieldsExposed, SIZE(allMut + externallyMutableFields +
        arrayFieldsExposed)>0 as isExtMutable
WHERE CASE WHEN isFieldType THEN isExtMutable ELSE NOT isExtMutable END
WITH field,enclosingType,publicGetters,
    EXTRACT(subtypeNameInfo IN FILTER(subtypeNameInfo IN COLLECT(
        [isFieldType,fieldTypeOrSubtype])
    WHERE NOT subtypeNameInfo[0])| subtypeNameInfo[1]) as immutableSubtypes,
    FILTER(subTypePair IN COLLECT([isFieldType, allMut +
        externallyMutableFields + arrayFieldsExposed])
    WHERE subTypePair[0])[0][1] as mutabilityInfo
WHERE NOT mutabilityInfo IS NULL
RETURN 'Warning[OBJ-56] Field ' + field.name + ' declared in line '  +
    field.lineNumber + ' in class ' + enclosingType.fullyQualifiedName +
    ' is not public, but it is exposed in public methods such as ' +
    publicGetters + '. The problem is that there is at least one member
    (like ' + mutabilityInfo + ') that can be accessed by a client to
    change the state of the field ' + field.name +
    CASE WHEN SIZE(immutableSubtypes)=0 THEN '. You should implement
    an appropriate inmutable subtype as a wrapper for your attribute,
    as you have not created any yet.'
    ELSE '. Remember to use an appropiate inmutable subtype (such as ' +
    immutableSubtypes + ') as a wrapper for your attribute.'
    END
```

B.13 NUM50-J

```
MATCH (varDec)-[:MODIFIED_BY | :HAS_VARIABLEDECL_INIT]
    ->(mod)-[:ASSIGNMENT_RHS | :INITIALIZATION_EXPR]->(rightSide)
WHERE varDec.actualType IN ['float', 'double']
OPTIONAL MATCH (binopR{actualType:'int'})<-[:BINOP_RHS]-(division)
    <-[:BINOP_LHS|:BINOP_RHS|:UNARY_ENCLOSES |:CONDITIONAL_EXPR_ELSE|
        :CONDITIONAL_EXPR_THEN *0..]-(rightSide),(division)-[:BINOP_LHS]
```

```
    ->({actualType:'int'})
WITH varDec, COLLECT(rightSide.actualType IN ['float','double'])
    as rightSidesAreFloat, FILTER(x IN COLLECT([rightSide,binopR])
WHERE NOT x[1] IS NULL AND x[0].operator='DIVIDE' ) as lines
WHERE  NOT ANY( x IN rightSidesAreFloat WHERE x) AND SIZE(lines)>0
RETURN 'Warning [CMU-NUM50] A truncated integer division was detected
    in line(s) ' + REDUCE(seed='',x IN lines | CASE WHEN seed
    CONTAINS (x[0].lineNumber + '') THEN seed ELSE seed +
    x[0].lineNumber + ',' END ) + ', assigned to variables of type float
    or double. If you want to make a float/double division and assign the
    result to the variable ' +  varDec.name + ', you must include an
    operand as float/double. Otherwise you can change the type of ' +
    varDec.name + ' from float/double to int, as it is never used to
    store an actual float/double value.'
```

Appendix C

Features of the homogeneous datasets

Tables C.1-C.5 show the different features used to build the homogeneous datasets. We do not include any feature that may depend on the size of the program. For example, Table C.5 does not include features such as the number of classes or interfaces. Their occurrence is considered, yet relative to the number of types used in the program.

Name	Description
Category	Syntactic category of the current node, given the abstract grammar for the Java language.
First, second and third child	Syntactic category of the corresponding child node.
Parent node	Syntactic category of the parent node.
Role	Role played by the current node in the structure of its parent node.
Height	Distance (number of edges) from the current node to the root node in the enclosing type (class, interface or enumeration).
Depth	Maximum distance (number of edges) of the longest path from the current node to a leaf node.

Table C.1: Feature abstractions used for statements.

Name	Description
Visibility	Public, protected, package or private.
IsAbstract, IsStatic, IsFinal	True or false.
ReturnsVoid, Overrides	True or false.
Number of parameters	Number of declared parameters.
Number of generics	Number of type parameters declared in a generic method.
Number of throws	Number of exceptions declared in the throws clause.
Number of annotations	Number of annotations declared for that method.
Number of statements	Number of statements used in the method body.
Number of local variables	Number of local variables declared.
Naming convention	Naming convention used for the method name (snake_case, upper, lower, camel_up or camel_down).
Main naming locals	Main naming convention used for local variables.

Table C.2: Feature abstractions used for methods.

Name	Description
Visibility	Public, protected, package or private.
IsDefined	Whether a value is defined in its declaration.
IsStatic, IsFinal	True or false.
Number of annotations	Number of annotations declared for that field.
Value	If any, category of the expression assigned in its definition.
Naming convention	Snake_case, upper, lower, camel_up or camel_down.

Table C.3: Feature abstractions used for fields.

Name	Description
Visibility	Public or package (non public).
Category	Class, interface or enumeration.
IsAbstract, IsStatic, IsFinal	True or false.
Extends	Whether the type extends another type.
Number of annotations	Number of annotations declared for that type.
Number of extends	Number of types that the current type extends (Java interfaces may extend any number of interfaces).
Number of implements	Number of interfaces implemented.
Number of generics	Number of type parameters declared in that generic type.
Number of methods	Number of methods declared for that type.
Number of overloaded	Number of overloaded methods declared for that type.
Number of constructors	Number of constructors implemented in that type.
Number of fields	Number of fields defined in that type.
Number of nested classes	Number of nested classes defined in that type.
Number of inner classes	Number of inner classes defined in that type.
Naming convention	Naming convention used for the type name (snake_case, upper, lower, camel_up or camel_down).

Table C.4: Feature abstractions used for types.

Name	Description
Class percentage	Percentage of classes (out of all the types) defined in that program.
Interface percentage	Percentage of interfaces (out of all the types) defined in that program.
Enum percentage	Percentage of enumerations (out of all the types) defined in that program.
Code in default package	Whether the program implements types in the default package.
Code in packages	Whether the program implements types inside packages.

Table C.5: Feature abstractions used for programs.

Appendix D

Publications

The research work of this PhD thesis has been published in different journals. The following publications were published during the development of such work:

- Oscar Rodriguez-Prieto, Alan Mycroft, Francisco Ortin. An Efficient and Scalable Platform for Java Source Code Analysis using Overlaid Graph Representations. *IEEE Access*, volume 8, pp. 1-22, May 2020, doi: 10.1109/ACCESS.2020.2987631 JCR Impact Factor 4.098 (Q1).
- Francisco Ortin, Oscar Rodriguez-Prieto, Nicolas Pascual, Miguel Garcia. Heterogeneous tree structure classification to label Java programmers according to their expertise level. *Future Generation Computer Systems*, volume 105, pp. 380-394, April 2020, doi: 10.1016/j.future.2019.12.016. JCR Impact Factor 5.768 (Q1).
- Oscar Rodriguez-Prieto, Francisco Ortin, Donna O'Shea. Efficient Runtime Aspect Weaving for Java Applications. *Information and Software Technology*, volume 100, pp. 73-86, August 2018, doi: 10.1016/j.infsof.2018.03.012. JCR Impact Factor 2.921 (Q1).
- Francisco Ortin, Javier Escalada, Oscar Rodriguez-Prieto. Big Code: New Opportunities for Improving Software Construction. *Journal of Software*, volume 11, issue 11, pp. 1083-1088, November 2016, doi: 10.17706/jsw.11.11.1083-1088. Scimago Impact Factor 0.556 (Q3).
- Oscar Rodriguez-Prieto, Lourdes Araujo, Juan Martinez-Romo. Discovering related scientific literature beyond semantic similarity: a new co-citation approach. *Scientometrics*, volume 120, issue 1, pp. 105-127, May 2019, doi: 10.1007/s11192-019-03125-9. JCR Impact Factor 2.770 (Q2).

References

[1] Andrew W Appel and Jens Palsberg. *Modern compiler implementation in Java*. Cambridge University Press, New York, NY, USA, 2 edition, 2003. 1, 19

[2] Raoul-Gabriel Urma and Alan 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*, PLATEAU '12, page 35–38, New York, NY, USA, 2012. Association for Computing Machinery. 1

[3] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010. 1

[4] Francisco Ortin, Javier Escalada, and Oscar Rodriguez-Prieto. Big Code: new opportunities for improving software construction. *Journal of Software*, 11(11):1083–1088, 2016. 1, 5

[5] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 457–466, New York, NY, USA, 2010. Association for Computing Machinery. 1

[6] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008. 1, 10

[7] Roman Ivanov. Checkstyle. `https://checkstyle.sourceforge.io`, 2020. 1

[8] Coverity. Coverity scan static analysis. `https://scan.coverity.com`, 2020. 1, 11

[9] Raoul Gabriel Urma and Alan Mycroft. Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming*, 97:127–134, January 2015. 2, 9, 17, 35

[10] Google. BigQuery. `https://cloud.google.com/bigquery`, 2020. 2, 43

[11] Raoul-Gabriel Urma. Wiggle. `https://github.com/raoulDoc/WiggleIndexer`, 2020. 2, 9

[12] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. 2, 14

[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. 2

[14] Svetoslav Karaivanov, Veselin Raychev, and Martin 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*, Onward! 2014, pages 173–184, New York, NY, USA, 2014. ACM. 2

[15] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 359–368, New York, NY, USA, 2012. ACM. 2, 14, 51

[16] Javier Escalada, Francisco Ortin, and Ted Scully. An Efficient Platform for the Automatic Extraction of Patterns in Native Code. *Scientific Programming*, 2017:1–16, 2017. 2, 14

[17] Francisco Ortin, Oscar Rodriguez-Prieto, Nicolas Pascual, and Miguel Garcia. Heterogeneous tree structure classification to label java programmers according to their expertise level. *Future Generation Computer Systems*, 105:380–394, April 2020. 2, 5, 6, 17, 49

[18] CMU SEI. Carnegie Mellon University, Software Engineering Institute, Java coding guidelines. https://wiki.sei.cmu.edu/confluence/display/java/Java+Coding+Guidelines, 2020. 3, 7, 17, 27, 29, 33, 73

[19] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery. 3, 18, 44

[20] Samy Abu-Naser. Predicting learners performance using artificial neural networks in linear programming intelligent tutoring system. *International Journal of Artificial Intelligence & Applications*, 3:65–73, 03 2012. 5, 12, 49

[21] Raoul Gabriel Urma. Programming language evolution. Technical Report UCAM-CL-TR-902, University of Cambridge, Computer Laboratory, February 2017. 9

[22] Semmle. CodeQL. https://semmle.com/codeql, 2020. 9, 36

[23] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1213–1216, New York, NY, USA, 2011. Association for Computing Machinery. 10, 44

[24] Malcolm Atkinson, David DeWitt, David Maier, François Bancilhon, Klaus Dittrich, and Stanley Zdonik. The object-oriented database system manifesto. In *Deductive and Object-Oriented Databases*, pages 223–240. Elsevier, 1990. 10

[25] Felix Dietze, Johannes Karoff, André Calero Valdez, Martina Ziefle, Christoph Greven, and Ulrik Schroeder. An Open-Source Object-Graph-Mapping Framework for Neo4j and Scala: Renesca. In Francesco Buccafurri, Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar Weippl, editors, *International Conference on Availability, Reliability, and Security (CD-ARES)*, volume LNCS-9817 of *Availability, Reliability, and Security in Information Systems*, pages 204–218, Salzburg, Austria, 2016. Springer International Publishing. 10, 44

[26] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In H. Conrad Cunningham, Paul Ruth, and Nicholas A. Kraft, editors, *ACM Southeast Regional Conference*, page 42. ACM, 2010. 10

[27] Nathan Hawes, Ben Barham, and Cristina 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*, pages 4:1–4:6, 2015. 10

[28] Oshini Goonetilleke, David Meibusch, and Ben 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*, pages 574–583, 2017. 10

[29] Tian Zhang, Minxue Pan, Jizhou Zhao, Yijun Yu, and Xuandong Li. An open framework for semantic code queries on heterogeneous repositories. In *2015 International Symposium on Theoretical Aspects of Software Engineering*, pages 39–46. IEEE, September 2015. 10

[30] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, January 1991. 10

[31] Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing (Expert's Voice in Open Source)*. Apress, oct 2010. 10

[32] Tian Zhang, Xiaomei Zheng, Yan Zhang, Jianhua Zhao, and Xuandong Li. A declarative approach for Java code instrumentation. *Software Quality Journal*, 23(1):143–170, September 2013. 10

[33] Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series, 1995. 11, 19, 56

[34] G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. Manning Publications, nov 2013. 11

[35] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, jul 1999. 11

[36] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 2008. 11

[37] PMD. PMD Source Code Analyzer Project. https://pmd.github.io, 2020. 11

[38] Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heui Seok Lim. Mining biometric data to predict programmer expertise and task difficulty. *Cluster Computing*, pages 1–11, 1 2017. 11

[39] S. Abu Naser, A. Ahmed, N. Al-Masri, and Y. Abu Sultan. Human computer interaction design of the LP-ITS: Linear programming intelligent tutoring systems. *International Journal of Artificial Intelligence & Applications*, 2(3):60–70, 2011. 12

[40] W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 150–159, Oct 2007. 12, 13

[41] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computing Programming*, 74(7):470–495, May 2009. 12

[42] J. Mayrand, C. Leblanc, and E.M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 244–253, 1996. 12

[43] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, November 1998. 12

[44] Axivion. Project Bauhaus, 2019. 12

[45] Vera Wahler, Dietmar Seipel, Jurgen Wolff v. Gudenberg, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *Proceedings of the Fourth IEEE International Workshop in Source Code*

Analysis and Manipulation, SCAM '04, pages 128–135, Washington, DC, USA, 2004. 12

[46] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Washington, DC, USA, 2006. 13

[47] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. 13

[48] Ahmad Taherkhani. Using decision tree classifiers in source code analysis to recognize algorithms. *The Computer Journal*, 54(11):1845–1860, November 2011. 13

[49] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. Learning to blame: localizing novice type errors with data-driven diagnosis. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA '17, pages 60:1–60:27. ACM, 2017. 13, 51, 55, 73

[50] Dor Levy and Lior Wolf. Learning to align the source code to the compiled object code. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2043–2051, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. 14

[51] Veselin Raychev, Martin Vechev, and Eran Yahav. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. 14

[52] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 472–483, New York, NY, USA, 2014. 14

[53] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications, 2018. 14

[54] Mingming Lu, Dingwu Tan, Naixue Xiong, Zailiang Chen, and Haifeng Li. Program classification using gated graph attention neural network for online programming service, 2019. 14

[55] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML, pages 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. 14

[56] Trevor Cohn. Efficient inference in large conditional random fields. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *European Conference on Machine Learning*, ECML, pages 606–613, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 14

[57] Hendrik Blockeel, Tijn Witsenburg, and Joost N. Kok. Graphs, hypergraphs, and inductive logic programming. In Paolo Frasconi, Kristian Kersting, and Koji Tsuda, editors, *Proceedings of the 5th International Workshop on Mining and Learning with Graphs*, MLG' 07, pages 93–96, 2007. 14

[58] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. Active inductive logic programming for code search. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE, pages 292–303, Piscataway, NJ, USA, 2019. IEEE Press. 14, 15

[59] Qiang Zeng, Jignesh M. Patel, and David Page. Quickfoil: Scalable inductive logic programming. *Proceedings of the VLDB Endowment*, 8(3):197–208, 2014. 15

[60] Michael Collins and Nigel Duffy. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 263–270, Stroudsburg, PA, USA, 2002. 15

[61] Deqiang Fu, Yanyan Xu, Haoran Yu, and Boyang Yang. WASTK: A Weighted Abstract Syntax Tree Kernel method for source code plagiarism detection. *Scientific Programming*, 2017:7809047:1–7809047:8, 2017. 15

[62] Konrad Rieck, Tammo Krueger, Ulf Brefeld, and Klaus-Robert Müller. Approximate tree kernels. *Journal of Machine Learning Research*, 11:555–580, 2010. 15

[63] Joshua Bloch. *Effective Java (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008. 17, 21, 33, 34, 71

[64] 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)*, pages 235–239. IEEE, 2000. 17

[65] Marko A. Rodriguez. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, page 1–10, New York, NY, USA, 2015. Association for Computing Machinery. 18

[66] Neo4j. The Neo4j traversal framework. `https://neo4j.com/docs/java-reference/current/tutorial-traversal`, 2020. 18

[67] Francisco Ortin, Daniel Zapico, and Juan Manuel Cueva. Design Patterns for Teaching Type Checking in a Compiler Construction Course. *IEEE Transactions on Education*, 50(3):273–283, 2007. 19, 20, 75

[68] Internet Engineering Task Force (IETF). JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties. https://jwt.io, 2020. 30

[69] Internet Engineering Task Force (IETF). JSON Web Token (JWT). https://tools.ietf.org/html/rfc7519, 2020. 30

[70] Internet Engineering Task Force (IETF). Hypertext Transfer Protocol (HTTP/1.1): Authentication (RFC 7235). https://tools.ietf.org/html/rfc7235#section-4.2, 2020. 30

[71] CMU CERT. Carnegie Mellon University, CERT Division, Software Engineering Institute. https://www.sei.cmu.edu/about/divisions/cert, 2020. 33

[72] Robert C. Seacord and Jason A. Rafail. Secure coding standards. In *Proceedings of the Static Analysis Summit, NIST Special Publication*, pages 13–17, 2006. 33

[73] Oracle. Java Platform Standard Edition 7 Documentation. https://docs.oracle.com/javase/7/docs, 2013. 33, 34, 35

[74] Klaus Havelund and Al Niessner. JPL Coding Standard, Version 1.1. https://www.havelund.com/Publications/JavaCodingStandard.pdf, 2010. 33, 34

[75] Oscar Rodriguez-Prieto and Francisco 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, 2020. 33, 35

[76] Oracle. Java Platform, Standard Edition API Specification. https://docs.oracle.com/javase/8/docs/api/index.html, 2020. 33, 34

[77] Kirk Knoernschild. *Java Design: Objects, UML, and Process*. Addison-Wesley Professional, Indianapolis, IN, USA, 2002. 34

[78] Andreas Sterbenz and Charlie Lai. Secure coding antipatterns: Avoiding vulnerabilities. In *JavaOne Conference*, 2006. 34

[79] Oracle. The Java Tutorials. https://docs.oracle.com/javase/tutorial, 2020. 34

[80] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in Action*. Manning Publications Co., USA, 1st edition, 2014. 36

[81] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented Queries on Relational Data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 36, 44

[82] Semmle. Variant Analysis. https://semmle.com/variant-analysis, 2020. 36

[83] CUP research group. GitHub Java corpus. http://groups.inf.ed.ac.uk/cup/javaGithub, 2020. 36

[84] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 281–293, New York, NY, USA, 2014. Association for Computing Machinery. 36

[85] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 254–265, New York, NY, USA, 2016. Association for Computing Machinery. 36

[86] Mathieu Goeminne and Tom 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)*, ICSME '15, page 551–555, USA, 2015. IEEE Computer Society. 36

[87] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007. 36, 37

[88] David J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005. 37

[89] Francisco Ortin, Miguel A. Labrador, and Jose M. Redondo. A hybrid class- and prototype-based object model to support language-neutral structural intercession. *Information and Software Technology*, 44(1):199–219, feb 2014. 37

[90] Microsoft. Windows management instrumentation. http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582(v=vs.85).aspx, 2015. 38

[91] GQL. Graph Query Language, GQL Standard. https://www.gqlstandards.org, 2020. 44

[92] Semmle. LGTM, continuous security analysis. https://lgtm.com, 2020. 46

[93] Francisco Ortin, Miguel A. Labrador, and Jose M. Redondo. A hybrid class- and prototype-based object model to support language-neutral structural intercession. *Information and Software Technology*, 56(2):199–219, February 2014. 46

[94] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in java. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):85:1–85:31, October 2017. 49

[95] Carnegie Mellon University, Software Engineering Institute. Java coding guidelines. https://wiki.sei.cmu.edu/confluence/display/java/Java+Coding+Guidelines, 2019. 49

[96] HongYun Cai, Vincent Wenchen Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30:1616–1637, 2018. 51, 72

[97] L. Rokach and O. Maimon. Top-down induction of decision trees classifiers - a survey. *IEEE Transactions on Systems, Man, and Cybernetics: Systems, Part C*, 35(4):476–487, November 2005. 51

[98] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013. 55

[99] Alex A. Freitas. On rule interestingness measures. In Roger Miles, Michael Moulton, and Max Bramer, editors, *Research and Development in Expert Systems XV*, pages 147–158, London, 1999. Springer London. 57

[100] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984. 60

[101] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition*. Academic Press, Inc., Orlando, FL, USA, 4th edition, 2008. 60

[102] Zuzana Reitermanová. Data splitting. In *Proceedings of the 19th Annual Conference of Doctoral Student*, WDS, pages 31–26, 2010. 61, 62

[103] Francisco Ortin. Heterogeneous tree structure classification to label Java programmers according to their expertise level (support material website). http://www.reflection.uniovi.es/bigcode/download/2019/fgcs, 2019. 64, 65

[104] Alexandru Niculescu-Mizil and Rich Caruana. Predicting good probabilities with supervised learning. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 625–632, New York, NY, USA, 2005. 65

[105] Jacob Bellamy-McIntyre. Modeling and querying versioned source code in rdf. In Aldo Gangemi, Anna Lisa Gentile, Andrea Giovanni Nuzzolese, Sebastian Rudolph, Maria Maleshkova, Heiko Paulheim, Jeff Z Pan, and Mehwish Alam, editors, *The Semantic Web: ESWC 2018 Satellite Events*, pages 251–261, Cham, 2018. Springer International Publishing. 72

[106] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4):81:1–81:37, July 2018. 72

[107] Sašo Džeroski. Multi-relational data mining: An introduction. *SIGKDD Explorations Newsletter*, 5(1):1–16, July 2003. 72

[108] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *SIGKDD Explorations*, 5(1):59–68, 2003. 72

[109] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996. 73

[110] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206, 1999. 73

[111] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 199–215. Springer, 2005. 73

[112] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common C language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017. 73

[113] Miwa Sasaki, Shinsuke Matsumoto, and Shinji Kusumoto. Integrating source code search into git client for effective retrieving of change history. In *Proceedings of the Workshop on Mining and Analyzing Interaction Histories*, MAINT, page 1–5, 2018. 73

[114] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in GitHub. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. 74

[115] Oscar Rodriguez-Prieto and Francisco Ortin. ProgQuery website. `https://github.com/OscarRodriguezPrieto/ProgQuery`, 2020. 75

[116] Oscar Rodriguez-Prieto, Alan Mycroft, and Francisco Ortin. An efficient and scalable platform for Java source code analysis using overlaid graph representations. (to be published), 2020. 75, 76