# Graph Representations used in the design of ProgQuery

## (Technical Report)

Oscar Rodriguez-Prieto, Francisco Ortin[*]

*University of Oviedo, Computer Science Department,*
*C/Federico Garcia Lorca 18, 33007, Oviedo, Spain*

## Abstract

This technical report is a support document for the article *Efficient and Scalable Platform for Java Source Code Analysis using Overlaid Graph Representations*, written by Oscar Rodriguez-Prieto, Alan Mycroft and Francisco Ortin.

*Keywords:* Code analysis, graph database, coding guidelines, declarative query language, program representation, Cypher, Java, Neo4j.

## 1. Introduction

ProgQuery is a platform to allow users to write their own Java program analyses in a declarative fashion, using graph representations [1]. We modify the Java compiler to compute seven syntactic and semantic representations, and store them in a Neo4j graph database [2]. Such representations are overlaid, meaning that syntactic and semantic nodes of the different graphs are interconnected to allow combining different kinds of information in the queries/analyses. In this technical report, we describe the ontology defined to represent syntactic and semantic information of Java programs. For more information about ProgQuery, please check [1].

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.

## 2. Nodes

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

[*] Corresponding author

*Email addresses:* `rodriguezoscar@uniovi.es` (Oscar Rodriguez-Prieto), `ortin@lsi.uniovi.es` (Francisco Ortin)

*URL:* `http://www.reflection.uniovi.es/ortin` (Francisco Ortin)

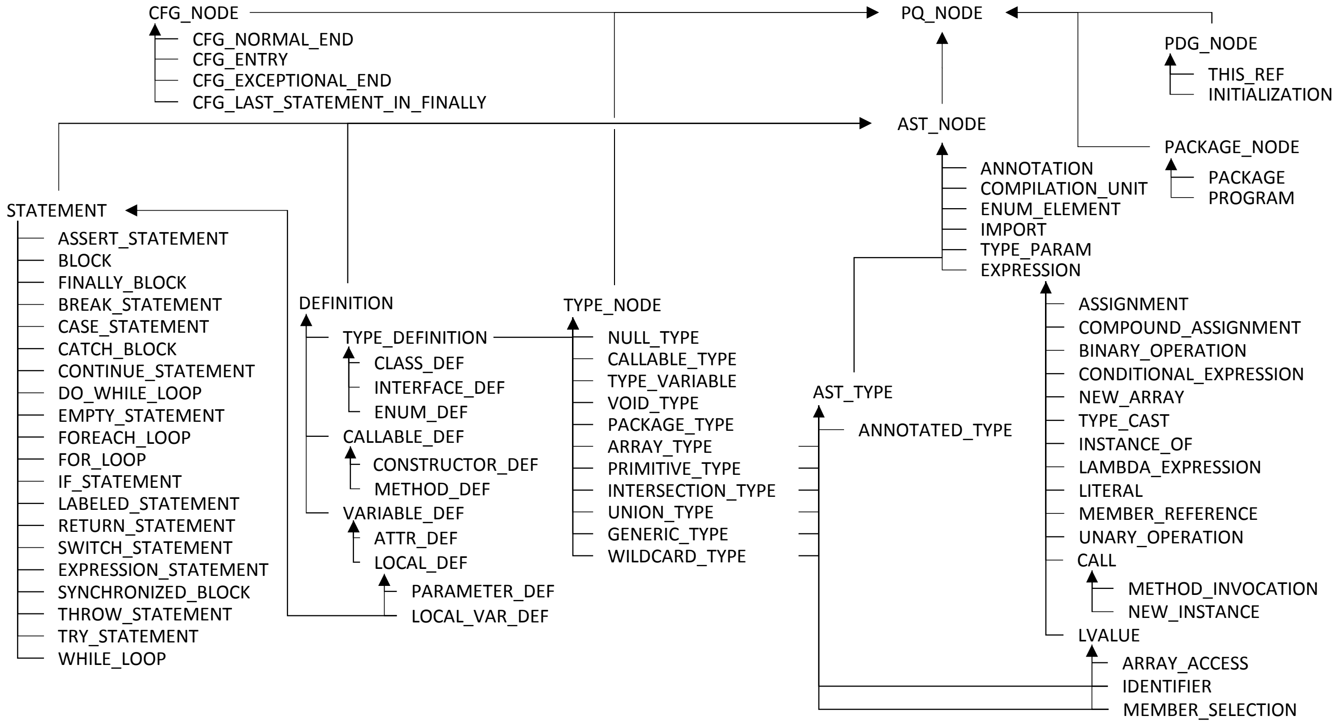

Figure 1: Labels defined to categorize the nodes used for the different Java program representations.

## 3. 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 1), 100 relationships and 26 properties. They define common syntax elements of an object-oriented language [3].

### *3.1. Nodes*

Root nodes in Figure 1 represent concrete nodes of the AST. When a program is represented, all the particular nodes in the AST are instances of these concrete labels. The rest of labels in Figure 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 [1]).
- `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 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 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 1).
  - MEMBER_SELECTION: represents expressions created with the . operator: field (obj.field) access, method invocation (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 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 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 1).

- AST_TYPE: generalization of types that could be writeN 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: intersection 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 conditions 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: extended 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 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 expression condition and the loop body.

*3.2. Relationships*

These are the relationships defined for the AST (their domain, range and cardinality are defined in Tables 1 and 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/increment 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 cases, 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 annotation 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 procedures:

- `database.procedures.getEnclosingClass`: relates a statement or variable definition to its enclosing class. Domain: STATEMENT $\cup$ 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 $\cup$ PARAMETER_DEF, range: CALLABLE_DEF, cardinality: 1.

| 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 | EXPRESSION | 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 | IDENTIFIER ∪ MEMBER SELECTION | 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 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 | 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 | 0..* |
| HAS_IMPLEMENTS_CLAUSE | CLASS_DEF ∪ ENUM_DEF | IDENTIFIER ∪ MEMBER_SELECTION | 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 − RECEIVER_PARAMETER | 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 2: Relatioships defined for ASTs (part 2).

- `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.

### *3.3. Properties*

The following properties were defined (detailed in Table 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`.
- `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.

## 4. Control Flow Graph

### *4.1. 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.

- `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.

### *4.2. Relationships*

We now describe the relationships of CFG. Table 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 an artificial statement representing the flow just before exiting the `finally` block.

- `CFG_NEXT_STATEMENT`: connects one statement with 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.

| 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 | 1 |
| paramIndex | Node | HAS_METHODDECL_PARAMETERS ∪ LAMBDA_EXPRESSION_PARAMETERS ∪ HAS_CLASS_TYPEPARAMETERS ∪ HAS_METHODDECL_PARAMETERS ∪ HAS_METHODDECL_TYPEPARAMETERS | Integer | 1 |

Table 3: Properties defined for ASTs.

- `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 the statement 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 procedures:

- `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..*.

### *4.3. Properties*

These are the properties of the CFG nodes and relationships (see details in Table 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.

| 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 4: Relationships defined for CFGs.

- `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.

| 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 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 6: Relationships defined for Call Graphs.

## 5. Call Graph

### *5.1. Nodes*

No new nodes are defined for the Call Graph.

### *5.2. Relationships*

These are the Call Graph relationships (detailed in Table 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.

### *5.3. 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 another initializer.

## 6. Type Graph

### *6.1. 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 an 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 7: Relationships defined for Type Graphs.

- TYPE_VARIABLE: type variables used with generic types.
- UNION_TYPE: union of two or more types, used in catch blocks (i.e., Java | type constructor).
- GENERIC_TYPE: a generic type that is parameterized with other types.
- WILDCARD_TYPE: node representing a Java wildcard type (i.e., ?).

### *6.2. Relationships*

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

- IS_SUBTYPE_EXTENDS: relates a type definition to its direct supertypes.
- IS_SUBTYPE_IMPLEMENTS: relates a class or enum definition to its direct super-interfaces.
- ITS_TYPE_IS: relates expressions, and variable and method/constructor definitions to their type.
- INHERITS_FIELD: relates type definitions to their (directly or indirectly) inherited fields, if any.
- INHERITS_METHOD: relates type definitions to their (directly or indirectly) inherited methods, provided that they are not overridden.
- OVERRIDES: relates a method definition to the overridden method definition, if any.
- ELEMENT_TYPE: relates an array type to the type of its elements.
- RETURN_TYPE: relates a callable type to its return type.
- PARAM_TYPE: relates a callable type to its parameter types, if any.

| 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 8: Properties defined for Type Graphs.

- 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*).

### *6.3. Properties*

The following properties are defined (Table 8):

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

## 7. Program Dependency Graph

### *7.1. 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 9: Relationships defined for PDGs.

### *7.2. Relationships*

Relationships defined for PDGs (Table 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).

### *7.3. 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).

## 8. 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..*.

## 9. Package Graph

### *9.1. 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.

| 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 10: Relationships defined for Package Graphs.

### *9.2. Relationships*

What follows are the Package Graph relationships defined (details in Table 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.

### *9.3. 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.

## Acknowledgments

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

## References

[1] O. Rodriguez-Prieto, A. Mycroft, F. Ortin, An efficient and scalable platform for Java source code analysis using overlaid graph representations, (to be published) (2020).

[2] O. Rodriguez-Prieto, F. Ortin, An efficient and scalable platform for Java source code analysis using overlaid graph representations (support material website), `http://www.reflection.uniovi.es/bigcode/download/2020/ieee-access` (2020).

[3] F. Ortin, D. Zapico, J. M. Cueva, Design Patterns for Teaching Type Checking in a Compiler Construction Course, IEEE Transactions on Education 50 (3) (2007) 273–283. doi:10.1109/TE.2007.901983.