Source-Code Queries with Graph Databases—with Application to Programming Language Usage and Evolution

Raoul-Gabriel Urma, Alan Mycroft

Computer Laboratory, University of Cambridge

Abstract

Program querying and analysis tools are of growing importance, and occur in two main variants. Firstly there are source-code query languages which help software engineers to explore a system, or to find code in need of refactoring as coding standards evolve. These also enable language designers to understand the practical uses of language features and idioms over a software corpus. Secondly there are program analysis tools in the style of Coverity which perform deeper program analysis searching for bugs as well as checking adherence to coding standards such as MISRA.

The former class are typically implemented on top of relational or deductive databases and make ad-hoc trade-offs between scalability and the amount of source-code detail held—with consequent limitations on the expressiveness of queries. The latter class are more commercially driven and involve more ad-hoc queries over program representations, nonetheless similar pressures encourage user-visible domain-specific languages to specify analyses.

We argue that a graph data model and associated query language provides a unifying conceptual model and gives efficient scalable implementation even when storing full source-code detail. It also supports *overlays* allowing a query DSL to pose queries at a mixture of syntax-tree, type, control-flow-graph or dataflow levels.

We describe a prototype source-code query system built on top of Neo4j using its Cypher graph query language; experiments show it scales to multi-million-line programs while also storing full source-code detail.

Keywords: Programming-Language Evolution, Source-Code Queries and DSLs, Graph Databases

On the occasion of his 65th birthday, we thank Paul Klint for his contributions to knowledge, for his role as co-founder and initial president of the *European Association of Programming Languages and Systems* (`eapls.org`), for his recommendations on software patents in Europe, and for generally being a good guy. Happy Birthday! Congratulations!

1. Introduction

Large programs are difficult to maintain. They are written by different software engineers in different styles. However, enhancements often require modifying several parts of the program at the same time. Hence, software engineers spend a large amount of time understanding the program they are working on.

Various automated tools have been developed to assist programmers to analyse their programs. For example, *code browsers* help program comprehension by hyperlinked code and by providing simple queries such as viewing the type hierarchy of a class, or finding the declaration of a method. These facilities are nowadays available in mainstream IDEs but are limited to fixed query forms and lack flexibility.

Several recent source-code querying tools provide flexible analysis of source code [12, 13, 1, 3, 16, 15]. They let programmers pose queries written in a domain-specific language to locate code of interest. For instance, they can be used to enforce coding standards (e.g. do not have empty blocks) [5], locate potential bugs (e.g. a method returning a reference type `Boolean` and that explicitly returns `null`) [2], code to refactor (e.g. use of deprecated features), or simply to explore the code base (e.g. method call hierarchy).

Email addresses: `raoul.urma@cl.cam.ac.uk` (Raoul-Gabriel Urma), `alan.mycroft@cl.cam.ac.uk` (Alan Mycroft)

Such tools also help programming language designers to analyse software corpora to learn whether a feature is prevalent enough to influence the evolution of the language. For example, a recent empirical study used a source-code query language to investigate the use of overloading in Java programs [14].

Typically, source-code querying tools use a relational or deductive database to store information about the program because of limitations on main memory. However, they often restrict queries to a subset of the source-code information (e.g. class and method declarations but not method bodies) because storing and exposing further information affects the scalability of the querying system. There is a trade-off: some systems scale to million-line programs but provide limited information (e.g. no method bodies) about the source code; others store more information about the source code but do not scale to large programs.

We argue not only that the *graph data model* conceptually unifies queries cross cutting over various representations of source code, but also demonstrate a prototype implementation based on Neo4j [6] which stores full source-code information and scales to programs with millions of lines of code.

Other authors have suggested using graphs in the context of software engineering research. For example, Nagl et al. describe the IPSEN system, a set of tools to facilitate the software development. It uses representations based on graphs. These graphs follow a predefined schema, which can be used to validate new specifications for a particular graph schema. Queries on graphs using subgraph matching are formulated in the PROGRESS language [19]. It provides the ability to locate specific nodes that meet given conditions and relationships between them. As a result, queries can be specified on software models that have graph-like structure. Moreover, researchers have also suggested using graph transformations for analysing refactoring dependencies and for specifying certain refactorings [18, 10].

Tools such as Findbugs and Coverity work at various levels of source-code representation to locate potential bugs or to enforce code standards. However, the trade-off is that programmers cannot easily specify ad-hoc code pattern queries via a DSL, instead having have to develop their own analyser calling an API provided by the tool. Nonetheless such a facility lets programmers build customised code analyses—which may be difficult or impossible on a source-code querying system which exposes less information.

1.1. Program Views and Overlays

At one level it seems 'obvious' that programs can be represented as a graph—a program is an abstract syntax tree which is a special case of an appropriately labelled graph. For example, a node may be labelled as being a `BinaryOp` and be connected via edges labelled `LEFT` and `RIGHT` to two other nodes. However, it is also 'obvious' that a program can be represented as a simple string, or as a control-flow graph. Some queries are more easily expressed in some representations but are less so, or impossible, to express in others. Lexical queries about comment placement may be impossible in a control-flow graph while queries about overloaded methods are problematic given a program represented as a string of tokens.

Many source-code queries which users might wish to make are cross-cutting between these levels. For example,"find a local variable which has 'gadget' as part of its name, which has type a subclass of 'Foo', which is declared in a method called recursively and which has multiple live ranges". One might even wish to query version history: "find methods that have been updated the most throughout the history of the software". Programming language designers might be interested in practical usage of Java covariant arrays which has inherently cross-cutting aspects [23].

We argue that while many source-code queries are cross-cutting, their individual components are framed in one of a small number of models of the program, such as tokens, abstract syntax trees, control-flow graphs, *already familiar to those familiar with compilers and related system tools*. We now identify these syntactic and semantic properties of programs which form bases for searches. We make these available as *overlays* in the graph model (see Section 2.2). These overlay properties can be seen as database *views*:

Textual/lexical. These queries express grep-like pattern-matching on the source as tokens or characters.

Structural. Structural queries operate on the abstract syntax tree (AST).

Semantic attributed. Structural queries *per se* can be clumsy: given a plain AST the query "find all overloaded methods". is very difficult to express. Indeed compiler-writers habitually decorate their syntax trees with additional information for later stages of the compiler to query. For example, an expression node representing a variable occurrence will contain a back-pointer to the node representing its *binding* declaration. Similarly, subclassing of type names will be represented directly as additional pointers. And of course, AST tree nodes for expressions will contain a pointer to their type. Such compiler back-pointers are exposed as overlay properties.

Control flow. Some queries are most easily expressed on a control-flow graph representation of the program, e.g. finding methods with particular forms of execution paths, or querying the call graph for interprocedural flow information.

Data flow. Often queries are best posed by reference to dataflow analysis results, e.g. live variable analysis and reaching definitions to find dataflow anomalies.

Run-time information. Profiling information can be useful to support querying memory and CPU consumption, as well as performance bottlenecks e.g. due to synchronized blocks.

Version history. These queries reference the version history of a code base, e.g. to identify methods introduced in the same commit [11]

Compiler diagnostics. While any warning generated by the compiler could, in principle, be reproduced by a suitable source-code query, at times of language evolution querying compiler warnings can be useful. There has been discussion about how to identify possible incompatibilities between Java 7 and Java 8 by inspecting the inference-resolution log generated by the Java compiler [4].

2. Graph Data Model

The graph data model encodes entities and relationships amongst them using a *directed graph structure* [8]. It consists of sets of nodes and binary edges (and hyperedges but we do not use these here) representing entities and relationships between these entities. Nodes and edges can be labelled, typically with the name of the entity they represent, but possibly with a tuple of name and further values, such as "weight:3". For our purpose, we will refer to such tuples as *properties* on nodes or edges.

As an example, a node may have a property `nodeType` with enumeration value `JCBinary` (names used in this and later examples are modelled on those in the Java compiler AST) to represent a binary expression. This node would typically be connected to two other nodes via edges labelled `LEFT` and `RIGHT`. By contrast, the *relational model* represents entities as tuples that are grouped into formal relations. Relationships between entities are encoded by combining relations based on a common attribute.

2.1. Graph Databases

Databases supporting the graph data model are often referred to as *graph databases*. They can be accessed via queries expressing graph operations such as traversal, pattern matching and graph metrics. Section 3 gives sample queries encoded in the *Cypher* graph query language used by the Neo4j graph database. By contrast databases supporting the relational model are referred to as *relational databases* and are typically accessed using SQL as the query language. There are two fundamental differences between relational and graph databases which we exploit in our work: *index-free adjacency* and *semi-structured data*.

First, graph databases provide index-free adjacency; i.e. records themselves contain direct pointers connected nodes. This obviates the need to combine relations based on a common attribute to retrieve connected records. Vicknair et al. [24] compared the performance of a graph database against a relational database; they reported that graph databases were faster when traversing connected data, sometimes by a factor of ten.

Second, relational databases depend on a schema to structure the data whereas graph databases typically do not have this requirement (i.e. they support *semi-structured data*). In practice, it means that in a graph database all records need to be examined to determine their structures, while this is known ahead of time in a relational database. This potential source of inefficiency is addressed in some graph databases which provide index facilities to map certain structures known ahead of time to a list of records. In addition, because graph databases do not depend on a schema, they can be more flexible when the structure of records needs to be changed. In fact, additional properties can be added to or removed from records seamlessly, which facilitates the addition of user-defined overlays by a DSL. In relational databases, schema modification is required before altering the record structure.

2.2. Overlays

The most obvious graph model of source code is the abstract syntax tree. However, as noted in Section 1.1 even finding the source-language type of a node representing a variable use may require an iterative search through scopes to find the *binding* declaration which contains the same variable name—and also its type.

What we need is an additional edge between a variable use and its binding declaration—this additional relation is an *overlay*. For example, we connect method and constructor invocations to their declarations with a labelled edge `HAS_DECLARATION`. Similarly we might want to pre-compute a program's call graph and store it as an overlay relation instead of searching, on demand, for call nodes within its body.

However, it is difficult to know ahead of time all possible information that users may want to query—consider querying for whether a given method is overloaded. This could be implemented directly as a query which processes all method-definition nodes in the program model, grouping them by fully qualified name and returning *true* if this group contains more than one member. However, this query would be more efficient if method nodes had a pre-computed (overlay) property indicating whether they are overloaded.

We return to this point in the next section but note that, as explained earlier, graph databases rely on semi-structured data so new overlays can be added without the need to change a schema. As a result, our model can easily cache results of queries as additional (overlay) relations or nodes.

2.3. Specifying and Implementing Overlays

At the moment, our prototype implementation stores as overlays those relations corresponding to compiler data structures (binding declarations, control-flow graph successors and predecessors, call graph and the like) *as the authors felt useful*. However, this is rather ad-hoc and we now turn to a more disciplined approach. Ideally, we would like overlays to be seen as cached results of queries—thus we *specify* the binding declaration of a variable use as the result of a search; this can be *implemented* either in terms of an additional pre-computation for every variable at start-up time, or in terms of a cached search result.

This exposes an expressiveness question: suppose we wish to provide overlay information for data-flow purposes (e.g. which assignments *reach* a given variable use, or which variables may alias at a given variable use). In general such data-flow analysis involves fixed-point computations; these are in general rather more complex than simple transitive closure which is often provided directly in query languages (e.g. Andersen's simple points-to analysis requires *dynamic* transitive closure). We plan to investigate whether the query language can be refined to permit direct representation of data-flow computations along the lines of [20]—thus crossing from Query Language to DSL.

2.4. Query Language vs. DSL

Query Languages sometimes solely permit simple queries, while others are more DSL-like (cf. Prolog which permits a sequence of function-like definitions prior to the final query)—for example transitive closure can often be expressed as a definition if it is absent from the core query language.

The definitions can then be implemented either as simple functions invoked during search (which could be slow if they are used many times), or as pre-calculating and storing overlay relations before the query(ies) themselves are evaluated—or even as demand-based caching.

3. Wiggle: a Prototype Graph-Model Code-Query System

We have built a prototype source-code querying system based on the graph data model, called *Wiggle* [7]. It consists of a Java compiler plug-in which traverses the Abstract Syntax Tree (AST) of each source-code file. It translates the AST elements and their properties to graph nodes in Neo4j. Links between AST elements are translated to labelled edges between these nodes. We describe our methodology for Java, however, it can be applied to other object-oriented programming languages that provide a standard AST representation.

The Java compiler defines various AST node types to represent different source-code elements. For example, a method declaration is represented by a class named `JCMethodDecl`, a wildcard by a class named `JCWildcard`. We map these AST nodes into nodes in our graph model and re-use their class names to identify the nodes. We store the names in a property called `nodeType` (an enumeration containing `JCMethodDecl` etc.) attached to each node. Some AST node have fields pointing to other AST nodes to represent parent-child relationships. For example, a

binary expression is represented by a JCBinary class, which contains fields named LHS and RHS pointing to its child expressions. We map these names to labelled edges. As an example, in our model JCBinary is mapped to a node with a property (nodeType:JCBinary), two edges labelled respectively LHS and RHS connecting two separate nodes with a property (nodeType:JCExpression). The complete list of relationships consists of around 120 different labelled edges and can be accessed on the project home page [7].

Wiggle retains the full AST and does not throw away any information; the prototype also implements overlays to access type hierarchy, type attribution and data-flow information. Its source code is available on the project homepage.

We now describe the pre-computed overlays in our model corresponding to the program views described in Section 1.1.

Type Hierarchy. We overlay the type hierarchy defined in the program on top of the AST. We connect nodes that represent a type declaration with a directed edge to show a type relation. These edges are labelled to distinguish between an IS_SUBTYPE_IMPLEMENTS or IS_SUBTYPE_EXTENDS relation. These overlays let us map the problem of finding the subtypes of a given type as a graph path-finding query. As a result, we do not need to examine class and interface declarations to extract the information. Note that we prefer to add overlays for one-step relations (e.g. direct subtype) as transitive closure is often provided directly by a graph query language—see the examples (in the Cypher query language of Neo4j) in the next section.

Override Hierarchy. Similarly, we overlay the override hierarchy of methods defined in the program on top of the AST. We connect method declaration nodes in the AST with the parent method they override using labelled edges OVERRIDES. As a result, we can query the override hierarchy directly instead of examining each class and method declaration to extract the information.

Type Attribution. We overlay a property called actualType on each expression node in the AST. It holds the string representation of the resolved type of an expression. This facilitates common source-code queries such as finding a method call on a receiver of a specific type. Indeed, this query would simply retrieve the method invocation nodes (labelled JCMethodInvocation) and check that the property actualType is equal to the desired type. Similarly, one can find covariant array assignments by comparing the actualType property of the left-hand and right-hand side of an assignment node (labelled JCAssign).

The Java compiler also defines an enumeration of different type kinds that an expression can have. These include ARRAY, DECLARED (a class or interface type), INT, LONG. We store these values in a typeKind property.

Method Call Graph. We construct the method call graph by extracting the method invocations from the body of each method declaration. Each method invocation is resolved and associated with its method declaration based on two filters: the static type of the receiver (weak call graph) and all possible subtypes of the receiver static type (rich call graph). We connect a method invocation node with its resolved method declaration(s) with a labelled edge CALLS for the weak call graph and CALLS_DYN for the rich call graph.

Typical source-code queries such as finding all the methods called within the entry point of a class can be seen as a traversal query on the generated call graph. In addition, we can easily find recursive methods by querying for method-declaration nodes with a CALLS cycle.

Data Flow. Dataflow analysis is a vast topic; this work describes only a simple set of data flow queries. We connect each expression with the declaration of the variables that they use using a GEN edge. Similarly, variables that are necessarily written to are connected to the expression with a KILLS edge. Such information is useful for performing certain data flow analysis such as live variable analysis; the definitions above are conservative for this—we consider the conservative set of variables that *must* be written to, but only these which *may* be read.

Next, we construct the read and write dependency graphs for each block. We inspect each block declarations (e.g method declarations, loops) to extract any field and variable accesses. We connect the blocks to the field or variable declarations that they read from or write to using READS and WRITES edges. The READS set of a block can be seen as the union of all the outgoing nodes connected with a GEN edge from all the expressions contained inside that block. Similarly, the WRITES set can be seen as the union of all outgoing nodes connected with a KILLS edge from all the expressions contained inside that block. This additional overlay is useful to easily compose program comprehension queries such as determining which fields are written to by a given method declaration.

3.1. Wiggle Examples

We show four examples of source code queries based on the graph data model for both program comprehension and language research. The queries are encoded in the Cypher language of Neo4j. The notation `-[:NAME]->` refers to an edge labelled `NAME` with `-[:NAME*]->` representing its transitive closure. Note the need for the ability to *bind* match positions to query-language variables for use in nested searches.

Query 1 shows how to find all classes that are directly or indirectly subtypes of `java.lang.Exception`. We generate a collection of all subgraphs starting at a node bound to the variable `n` that is connected to a node `m` that represents `java.lang.Exception`. Note the use of transitive closure to specify a path reaching `m` of arbitrary length as long as it solely consists edges labelled `IS_SUBTYPE_EXTENDS`. Finally, we bind all the subgraphs that were matched into a variable `path`, which we return.

Query 2 searches for classes containing recursive methods. First, we look up the index of nodes to find all method nodes that have the property `nodeType` equal to `JCMethodDecl` and iteratively bind them to the variable `m`. Next we specify a graph pattern using the `MATCH` clause that generates a collection of all subgraphs starting at a node `c` (a class declaration) that is connected to `m`. In addition, we specify that the method `m` is calling itself by specifying an edge `CALLS` from `m` to `m` (weak call graph).

Query 3 reports a ratio of the usage of generic wildcards. Cypher provides aggregate functions such as `count(*)` to count the number of matches to a grouping key. The `RETURN` clause uses this feature to count the number of occurrences based on `typeBoundKind` property of nodes of type `JCWildcard`.

Query 4 finds all uses of covariant array assignments. First, we look up nodes of type `JCAssign`, which represent an assignment in the AST. We iteratively bind them to the variable `n`. Next, a `MATCH` clause decomposes an assignment into its left and right children, which we bind respectively to the `lhs` and `rhs` variables. We ensure that these nodes are both tagged as holding an expression of type `ARRAY` by restricting the `typeKind` property using a `WHERE` clause. Finally, we also ensure that the resolved type (e.g. `Object[]`) and `String[]` of the left- and right-hand sides of the assignment are different by checking the `actualType` property, which holds the string representation of the type of expression.

Query 1

```
START m=node:node_auto_index(nodeType='ClassType')
MATCH path=n-[r:IS_SUBTYPE_EXTENDS*]->m
WHERE m.fullyQualifiedName='java.lang.Exception'
RETURN path;
```

Query 2

```
START m=node:node_auto_index(nodeType='JCMethodDecl')
MATCH c-[:DECLARES_METHOD]->m-[:CALLS]->m
RETURN c,m;
```

Query 3

```
START n=node:node_auto_index(nodeType ='JCWildcard')
RETURN n.typeBoundKind , count(*)
```

Query 4

```
START n=node:node_auto_index(nodeType='JCAssign')
MATCH lhs<-[:ASSIGNMENT_LHS]-n-[:ASSIGNMENT_RHS]->rhs
WHERE has(lhs.typeKind) AND lhs.typeKind='ARRAY' AND
      has(rhs.typeKind) AND rhs.typeKind='ARRAY' AND
      lhs.actualType <> rhs.actualType
RETURN n;
```

3.2. Wiggle Performance

We evaluated our system by executing various queries for program comprehension and language design research on a corpus of 12 Java projects (*Apache Ant*, *Apache Ivy*, *AoI*, *HsqlDB*, *jEdit*, *Jspwiki*, *Junit*, *Lucene*, *POI*, *Voldemort*, *Vuze* and *Weka*) totalling *2.04 million lines of code*. We report the response time for the queries described in the previous section when executed on a typical personal computer used for programming tasks: a MacBook Pro with 2.7GHz Intel Core i7, 8GB of memory and running OSX 10.8.2. We used Neo4j standalone server 1.9.M04 and OpenJDK 1.7.0. Pre-processing the source code and loading into Neo4j as a transaction took between two and ten times longer than compiling it using *javac*. Our current implementation processes each compilation unit as one database transaction—we believe these times could be improved by batching the loads.

For query execution time, we distinguish between a *cold* system (i.e. reporting the response time of a query after restarting the querying system) and a *warm* system (i.e. we ran the same query twice and reported the faster response time) to investigate the impact of caching on performance. In practice developers often execute multiple

queries having common sub-queries as well as repeating near-identical queries. To aid performance a production source-querying system could warm the cache ahead of time with 'anticipated' queries.

With a cold cache, Query 1 ran in two seconds, Query 2 took twelve seconds and Query 4 in less than ten seconds. Query 3 took less than half a second. This shows that while our system can efficiently traverse graph structures, aggregate operations are also efficiently executed. We found that running queries with a hot cache decreases response time by a factor of 3x to 5x compared to a cold cache.

Finally, we ran the same queries with program of increasing size (from three thousand to two million lines of code) and found that the response times scaled linearly with increasing program size.

3.3. Comparison with Other Systems

We provide a comparative study of related systems as to their power to express different queries in previous work [22]. Unfortunately, while JQuery is easily accessible, we could not find a working implementation for recent systems such as CodeQuest and JTL. However, the authors reported the absolute times of various source-code queries executed on their system, which enables crude comparison. Note that it is difficult to make a fair comparison since the queries were executed on older desktop machines than the laptop we used for our experiments. On the other hand, our system stores more program source-code information; CodeQuest, JTL and JQuery do not support method bodies, generics or types of expressions.

JQuery. We created three queries in JQuery: finding recursive methods, classes declaring public fields and overloaded methods. We tested them on the *Ivy* program (68k LoC) because JQuery failed indexing larger projects such as *HsqlDB*. This deficiency was already reported in other work [12, 15]. All queries executed in a couple of seconds except overloaded methods which took somewhat over 30s on our MacBook machine.

JTL. The authors reported execution times between *10s* and *18s* for code exploration queries carried out on *12,000* random classes taken from the Java standard library. However, it is unclear whether the experiment was performed with a cold or warm system. We obtained times of 0s to 12s running on a *cold* system and 0s to 3s with a *warm* system by running queries on a larger data set (a Corpus of over *13,000* Java source files and 2.04 million lines of code). Note that our setup is given an advantage because the JTL experiments were reported on a 3GHz Pentium 4 processor with 3GB of memory whereas we used a more modern machine (MacBook Pro with 2.7GHz Intel Core i7 and 8GB of memory).

CodeQuest. The authors ran various queries on different database systems. We compare with the fastest times they report, which were found on a quad Xeon 3.2GHz CPU with 4GB of memory and using XSB (a state of the art deductive database). The authors reported times between *0.1s* and *100s*. The slowest query is looking for methods overriding a parent method defined as `abstract`. In the CodeQuest system, the overriding relation between methods is computed on the fly. In our system, the 'overrides' relation is already pre-computed as an *overlay* on the AST.

4. Programming Language Evolution

4.1. Searching

Empirical studies help to understand how programming language features or idioms are used in practice. They answer questions and hypotheses that can influence the evolution of programming languages. There are several proposals for open-source corpora such as Qualitas Corpus that aim to help language researchers examine code [21]. Even so, conducting studies can be difficult, time-consuming and hard-to-replicate by the research community. For example, analysing the usage frequency of a specific idiom requires complex static analysis and reporting tools—frequently specially built and mutually incompatible. Source-code querying infrastructures such as Wiggle ease the process: researchers express a research question as a source-code query over a corpus of software. However, as noted earlier, systems prior to Wiggle make ad-hoc trade-offs between scalability, amount of source-code detail held in the database, and expressiveness of queries.

Systems like Wiggle let language designers rapidly test programming language evolution hypotheses by searching complex code patterns and gain insight into what programmers do in practice.

A well-known example of language usage driving language evolution is that of the `for` loop to iterate over arrays. Programmers traditionally used it to initialise and to increment an index variable used to access array elements. However, this approach is error-prone: should we start the iteration at index 0? Should we stop iterating before or after we reach the size of the array? As a result, many programming languages introduced the `foreach` construct, which abstracts away the iteration logic. The benefit is that programmers now make fewer mistakes and their code becomes more readable.

Querying for such a code pattern needs various cross-cutting information: *structural* to identify use of for loops, *type information* to identify numerical incremented variables, *data-flow* information to ensure that the index variable used to access array elements is not written to.

Queries in Wiggle are currently expressed via Cypher. However, certain queries such as finding covariant array assignments (Query 4 above) can be clumsy to write—queries involving several steps through a graph are expressed as conjunctions of single-step primitives. Hence, we are also designing a user-friendlier source-code query language (inspired by Query By Example [25] and SQL forms) which compiles to Cypher but which is easier and more intuitive to use. The idea of Query by Example is that queries are written as source-language phrases containing variables prefixed by underscore, thus replacing explicit decomposition with pattern matching. Query 4 (finding covariant assignments) could then be expressed as:

```
JCExpression(_E1 = _E2); actualType(_E1, _T1[]); actualType(_E2, _T2[]);
```

along with "`_T1 <> _T2`" in the the Query-by-Example *Condition Box* where semantic constraints (those not captured by pattern matching) are specified. Note that, while we logically require "`_T2 <: _T1`" (subtyping) for covariance, this is equivalent to "`_T1 <> _T2`" (inequality) for Java programs which have already successfully passed type-checking.

4.2. Replacing

Another aspect of programming language evolution is to help programmers migrate between different versions of a language or libraries [9]. This requires the ability to specify transformations between code patterns; for example transforming all `for` loops to iterate over arrays to using the `foreach` loop. There are many similar examples such as replacing covariant-array uses with generics, or even syntax incompatibilities between Python 2 and Python 3.

We are currently extending Wiggle so language designers can specify *code transformations*. This is achieved by specifying (conditional) rewriting rules on the graph model of the source code: a subgraph replaced with another subgraph provided some condition holds (e.g. no edge stating that a variable node is written to inside the body of a for loop). Mens et al. previously suggested using graph productions to formalise certain refactorings such as *pull up method* and *encapsulate a variable* [17]. Building on this work, we are investigating how to extend the Cypher query language to provide graph-production primitives to express transformation between AST patterns. Such an extension would provide a means for language designers or programmers to search for code patterns and to automatically upgrade code bases with newer equivalents.

This "search and replace" metaphor for code queries exactly parallels that of text editors.

5. Conclusions and Further Work

We presented the design and implementation of a source-code querying system using a *graph database* that stores full source-code detail and scales to program with millions of lines of code. It is based on the concept of the *graph data model*, not only representing the AST itself, but also various additional relations which can simplify queries and speed up their implementation via the concept of *overlays*. This allows queries to be posed at a mixture of syntax-tree, type, control-flow-graph and dataflow levels.

We described Wiggle, our prototype implementation of source-code queries using the graph data model. We showed it can naturally express various source-code queries (encoded as Cypher) with examples of code exploration and language research queries. Performance evaluation for programs and corpora of a range of sizes suggests that existing graph databases can efficiently query programs with millions of lines of code whilst retaining full source-code detail.

The most important future work is to generalise the current fixed overlay structure to allow user-specified overlays by an enhanced DSL—and determining what features this DSL requires. Another issue is query optimisation, or warning of high-complexity queries (for large corpora this perhaps means "worse than $O(n)$"). We are also investigating how to extend the graph query language with primitives to specify graph transformations, which will result in code transformations. An attractive possibility is that new language versions come with a script to transform older features instead of merely 'deprecating' them.

Acknowledgements

We thank Alex Buckley, Joel Borggrén-Franck, Jonathan Gibbons and Steffen Lösch for helpful comments. We are grateful to the anonymous referees for their comments, particularly for pointing out Nagl's volume on the IPSEN project [19]. This work was supported by a Qualcomm PhD studentship.

References

[1] BBQ. `http://browsebyquery.sourceforge.net/`.
[2] Findbugs. `http://findbugs.sourceforge.net`.
[3] Jackpot. `http://wiki.netbeans.org/Jackpot`.
[4] Java 8: support for more aggressive type-inference. `http://mail.openjdk.java.net/pipermail/lambda-dev/2012-August/005357.html`.
[5] JPL institutional coding standard for the Java programming language. `http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_Java.pdf`.
[6] Neo4j graph database. `http://www.neo4j.org/`.
[7] Wiggle project. `http://www.urma.com/wiggle`.
[8] R. Angles and C. Gutierrez. Survey of graph database models. *Computing Surveys*, 40(1):1, 2008.
[9] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA*, pages 265–279. ACM, 2005.
[10] P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Coordinated distributed diagram transformation for software evolution. *Electronic Notes in Theoretical Computer Science*, 72(4):59–70, 2003.
[11] S. Breu and T. Zimmermann. Mining aspects from version history. In S. Uchitel and S. Easterbrook, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. ACM Press, September 2006.
[12] T. Cohen, J. Y. Gil, and I. Maman. JTL: The Java tools language. In *OOPSLA*, 2006.
[13] C. de Roover, C. Noguera, A. Kellens, and V. Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *PPPJ*, 2011.
[14] J. Gil and K. Lenz. The use of overloading in Java programs. In *ECOOP*, 2010.
[15] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with Datalog. *ECOOP 2006 – Object-Oriented Programming*, pages 2–27, 2006.
[16] D. Janzen and K. de Volder. Navigating and querying code without getting lost. In *AOSD*, 2003.
[17] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.
[18] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software & Systems Modeling*, 6(3): 269–285, 2007.
[19] M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*, 1996. Springer. ISBN 3-540-61985-2.
[20] A. Stone, M. Strout, and S. Behere. May/must analysis and the DFAGen data-flow analysis generator. *Information and Software Technology*, 51(10):1440–1453, 2009.
[21] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336–345. IEEE, 2010.
[22] R.-G. Urma and A. Mycroft. Programming language evolution via source code query languages. In *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools (PLATEAU)*, pages 35–38. ACM, 2012.
[23] R.-G. Urma and J. Voigt. Using the OpenJDK to investigate covariance in Java. *Oracle Java Magazine*, May 2012.
[24] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, page 42. ACM, 2010.
[25] M. M. Zloof. Query by example. In *AFIPS National Computer Conference*, pages 431–438. AFIPS Press, 1975.