Science of Computer Programming 97 (2015) 127–134

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Source-code queries with graph databases—with application to programming language usage and evolution

Raoul-Gabriel Urma, Alan Mycroft

Computer Laboratory, University of Cambridge, United Kingdom

HIGHLIGHTS

- A scalable source-code querying system which stores full source-code detail.
- Queries on syntax, types and data flow via overlay relations (database views).
- Based on the graph data model, with prototype ('Wiggle') using Neo4j.

ARTICLE INFO

Article history:
Received 30 September 2013
Accepted 5 November 2013
Available online 12 November 2013

Keywords:
Programming language evolution
Source-code queries and DSLs
Graph databases

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 data-flow 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.

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!

E-mail addresses: raoul.urma@cl.cam.ac.uk (R.-G. Urma), alan.mycroft@cl.cam.ac.uk (A. Mycroft).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.010

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).

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 affect 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).