Query Technologies and Applications for Program Comprehension

In Proceedings of ICPC 2008, pp. 285-288

Mathieu Verbaere
Semmle Limited
United Kingdom

Michael W. Godfrey
University of Waterloo
Canada

Tudor Gîrba
University of Berne
Switzerland

Abstract

*Industrial software systems are large and complex, both in terms of the software entities and their relationships. Consequently, understanding how a software system works requires the ability to pose queries over the design-level entities of the system. Traditionally, this task has been supported by simple tools (*e.g., grep*) combined with the programmer's intuition and experience. Recently, however, specialized code query technologies have matured to the point where they can be used in industrial situations, providing more intelligent, timely, and efficient responses to developer queries.*

This working session aims to explore the state of the art in code query technologies, and discover new ways in which these technologies may be useful in program comprehension. The session brings together researchers and practitioners. We survey existing techniques and applications, trying to understand the strengths and weaknesses of the various approaches, and sketch out new frontiers that hold promise.

1 Introduction

Understanding source code is vital to many tasks in software engineering. Developers need to understand the system they work on in order to improve quality, enhance functionality, and re-engineer for new deployment environments. Querying tools are designed to help developers comprehend the varied and complex relationships within their codebase. They enable the interactive exploration of a system and the definition of custom views.

Recently, the use of code queries has spread to other tasks, *e.g.*, for detecting architectural flaws, identifying refactoring opportunities, locating project-specific bugs, finding crosscutting concerns. Queries may also be used in a diagnostic role, such as for monitoring development progress, assessing design quality, tracking unwanted dependencies, and gathering quality metrics on every build.

For queries to be effective, we need to integrate them with the rest of software engineering tools. At the very least, query results must be presented to the developer. One direction is to provide visualization techniques that correlate the results of the queries with the overall structure of the codebase under study. Another option is to take query results as input for further queries and analyses.

This short paper is a brief introduction to code queries and their applications. First, we describe the use of queries in two major areas of software engineering: quality assurance and reengineering. Then, we highlight the important role of queries in software visualization, and present different code query technologies. Finally, we discuss some challenges around code queries to foster the initial discussion of the session.

2 Queries for quality assurance

Software quality can be assessed and improved by computing metrics, finding common bugs, checking style rules and enforcing coding conventions that are specific to an API.

A wealth of different metrics have been proposed to detect design flaws [20]. Metrics values are usually presented in isolation or grouped to measure a particular design aspect, but design problems are difficult to spot from a large set of abnormal values. By allowing metrics results to be filtered and tuned to the specificities of a system, queries can help developers and architects pin down the real cause of design flaws [23].

More and more development teams now run static program analyzers on a daily basis to check for common bugs. When expressed as queries, these checks can easily be tuned per project to reduce the number of false positives. Perhaps where the greatest benefit lies, however, is in framework-specific checks. Indeed, modern systems rely on an ever growing number of frameworks. Queries allow the expression of concise, executable rules that client code of an API must follow [5].

Naturally, quality checks can be performed on different versions of the same system, to monitor its development. Queries are helpful for that purpose too. In particular, they can be used to check that no design inconsistencies are introduced during the evolution of a system [24].

3 Queries for reengineering

Not only can queries detect violations of certain design principles, but they also play an important role in driving development and performing design recovery. The canonical use of querying for such tasks involves several steps. First, the source code is run through a fact extractor, which creates a somewhat higher level model of the source code entities and their interdependencies than the code itself. Entities are commonly at the level of files, classes, functions/methods, global and instance variables, while relationships include file inclusion, calls, variable set/reference, and instantiation.

Once these models have been created, queries can be run to infer other relationships, check for design violations, and create higher level (package, subsystem) entities whose properties are based on those of their contained elements. For example, one might wish to know if there are any calls from subsystem A to B that do not go through module M (the intended interface for the subsystem). Or one might assign classes to subsystems and then verify that the subsystems form a layered architecture with no calls between modules that violate the layering.

The Grok query language has been used together with other tools such as fact extractors, for both reverse engineering of designs from code as well as for ensuring that architectural rules are enforced by developers [11].

A main concern in reengineering is to bridge the gap between models and source code. Reflexion models address exactly that issue by defining a mapping between the two [18, 27]. There again, mappings can be declared and checked with code queries.

4 Queries and visualization

Typically, we need to relate the results of a query to the context in which they live. Thus, visualization can complement queries in at least two ways:

- By locating the results in the overall system. For example, Seesoft is a tool for visualizing line oriented software information for several files [9]. Such a visualization offers a map of the whole system on which we can highlight the results of queries.

- By providing details about the results so that further analyses can be more efficient. For example, CodeCrawler [19] implements the concept of polymetric views to map metrics and design flaws on source code visualizations [20].

Other tools have been proposed to deal with the above issues. Rigi uses a graph metaphor for displaying software systems [26]. SHriMP provides an interactive environment in which the engineer can navigate the system [32]. NDepend highlights the results of queries on treemaps [28].

Visualization transforms the system models into graphical models, which are typically graphs. G^{see} provides a simple interface for specifying the mapping to visualization [10]. Mondrian also offers an engine for scripting such transformations [25]. These approaches aim at reducing the cost of building custom visualization, and as such they can be used for showing the results of custom queries.

5 Query technologies

There are many different approaches to querying source code. Perhaps the most natural solution is to store the program in a database. One of the earliest proposals of this kind was Linton's Omega system [21]; queries were formulated in the Ingres query language QUEL. In the same vein, the C Information Abstraction system [3] showed how to store the code structure of a system into a database for generating graphical views, extracting subsystems, and eliminating dead code. These systems have had quite an impact on industrial practice, as numerous companies now use a database system as a code repository, *e.g.* [2, 31]. Another natural possibility, given the tree structure of source code, is to employ an XML representation of the source code. In Sextant [7] for instance, software engineering tasks are expressed with XQuery. Of course, the clear advantage of XQuery and SQL is that they are already familiar to most programmers. Both languages, however, are quite verbose and less declarative for expressing recursive queries on the complex graph representation of a program. A typical example of such recursive queries is to search, while cleaning up a piece of legacy software, for methods that are never called directly or indirectly from *main* methods.

Logic programming languages are much better suited for those tasks. The XL C++ Browser [15] was one of the first to promote Prolog for expressing typical queries over source code. More recently, logic programming in the tradition of Prolog has inspired tools like Soul [24], JQuery [13] and JTransformer [17]. Yet, the learning curve of Prolog is quite long and complex queries are also often verbose.

In many systems therefore, a deliberate choice was made to simplify the set of expressible recursive queries. GraphLog [4] is a query language with just enough power to express paths queries on graphs. GReQL [6] is another graph query language, but with a syntax close to SQL. Further examples of domain-specific languages for code search are the program query language PQL [14] or pattern-based languages as enabled by SCRUPLE [29]. A few other systems went the neat route of relational algebra, *e.g.* Grok [11], CrocoPat [1] and RScript [16]. Perhaps the principal benefit of those systems is in efficient algorithms for implementing and optimizing relational operations — for instance, CrocoPat uses BDDs to compactly represent huge relations. Those systems, however, are usually not restricted to declarative queries; most of them embed imperative constructs to allow the extraction and manipulation of facts.

Interestingly, there is a language that is purely declarative, rule-based like Prolog, and that can be executed using relational algebra. That is Datalog, a query language originally put forward in the context of deductive databases. Datalog has been proposed at various occasions for analyzing source code, *e.g.* [4, 30]. It has a simple semantics and can be executed efficiently on top of a traditional relational database. This is the technique used in SemmleCode [31] where queries are written in .QL [5], an object-oriented query language with a syntax familiar to SQL and OO developers. For their execution, .QL queries are translated to a variant of Datalog, optimized and further compiled to SQL.

6 Open questions and challenges

Although code queries have been long and well studied (our brief survey above is far from being exhaustive), there still remain open questions and ongoing research challenges. We state here a few problems which we believe deserve clearer answers.

Heterogeneous models It appears now that there is no silver-bullet model for all applications in program comprehension. The finer a model is, the more interesting queries can be run, but the more complex it is to express such code queries. One challenge is to come up with a model that matches the mental representation that people make of a program. How do we address the fact that such mental representation naturally varies with the skills of the programmer and with the nature of the tasks? In addition, modern software systems are now composed from pieces of code written in a variety of programming languages, ranging from dynamic languages to statically-typed OO languages and the embedding of domain-specific constructs. What are the best ways to support queries that transcend language barriers?

Pattern-based *vs* general-purpose languages Typically, code query languages fall into one of two very different categories: pattern-based languages for querying code written in a specific object language and general-purpose languages for querying any kind of source code artifact. The former approach enables queries in concrete syntax, which clearly facilitates the adoption of queries (at least for simple tasks). On the other hand, general-purpose query languages seem to scale to more complex problems. What are the best ways of adding elements of concrete syntax to a general-purpose query language?

Mixing implementation strategies Here again, certain implementation strategies are better suited to certain contexts and tasks. For instance, using a database is crucial when querying very large systems, but slightly better performance can be achieved on medium-sized projects by fitting the whole codebase in memory. Another example is when querying for the transitive closure of a large call graph. The resulting relation is often too huge to be materialized on disk in reasonable time, whereas it could be compressed in a BDD as in [1]. Although query languages like PQL [14] and .QL [5] were designed so as to separate the language from its implementation details, there has been little work so far on mixing different evaluation strategies.

IDE integration The trend is now to integrate querying tools into IDEs, *e.g.* [13, 7, 17, 31]. Queries are run in an interactive setting where the codebase changes frequently. To illustrate, one may run a query to spot a refactoring opportunity, apply the refactoring, run another query, and so on. Also, lightweight static checks should be performed after each change, however minor it is. These querying scenarios present two challenges. First, the model representing the codebase has to be synchronized with the source code. Second, queries need to be rerun quickly and seamlessly, *i.e.* incrementally as in [8].

Code queries for other tasks Code queries have been proposed to navigate and visualize software systems. They have been used for the flexible definitions of code metrics, *e.g.* [23, 5]. They can also express complex custom code analyses to enforce project-specific design choice and coding conventions. We believe, however, that we have only scratched the surface of potential applications. For instance, code queries have been proposed to discover and document cross-cutting concerns [22]. It would be interesting to exploit further code queries as lightweight contract definitions to enforce documentation.

Querying all development artifacts Queries should not be limited to source code, but run on all other development artifacts: bug reports, version histories, documentation. Exciting work on mining software archives has shown how queries in such a broader context often reveal hidden but crucial information about the past, present and future of a project and its development team [12]. Surely, this work shall benefit from more accurate, more semantic queries that code querying tools enable.

References

[1] D. Beyer. Relational programming with crocopat. In *Proceedings of the 28th international conference on Software engineering (ICSE)*, pages 807–810. ACM, 2006.

[2] Cast. Company website at: `http://www.castsoftware.com`.

[3] Y. Chen, M. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, 1990.

[4] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proceedings of the 14th international conference on Software engineering (ICSE)*, pages 138–156, 1992.

[5] O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble. .QL for source code analysis. In *Source Code Analysis and Manipulation (SCAM)*, 2007.

[6] J. Ebert, B. Kullbach, and A. Winter. Querying as an enabling technology in software reengineering. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering (CSMR)*, page 42. IEEE Computer Society, 1999.

[7] M. Eichberg, M. Haupt, M. Mezini, and T. Schäfer. Comprehensive software understanding with Sextant. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 315–324, 2005.

[8] M. Eichberg, M. Kahl, D. Saha, M. Mezini, and K. Ostermann. Automatic incrementalization of prolog based static analyses. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 109–123, 2007.

[9] S. G. Eick, J. L. Steffen, and E. E. Sumner. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov. 1992.

[10] J.-M. Favre. Gsee: a generic software exploration environment. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 233–244. IEEE, May 2001.

[11] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[12] A. E. Hassan, A. Mockus, R. C. Holt, and P. M. Johnson. Guest editor's introduction: Special issue on mining software repositories. *IEEE Transactions on Software Engineering*, 31(6):426–428, 2005.

[13] D. Janzen and K. de Volder. Navigating and querying code without getting lost. In *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187, 2003.

[14] S. Jarzabek. Design of flexible static program analyzers with PQL. *IEEE Transactions on Software Engineering*, 24(3):197–215, 1998.

[15] S. Javey, K. Mitsui, H. Nakamura, T. Ohira, K. Yasuda, K. Kuse, T. Kamimura, and R. Helm. Architecture of the XL C++ browser. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 369–379, 1992.

[16] P. Klint. A tutorial introduction to RScript. Centrum voor Wiskunde en Informatica, draft, 2005.

[17] G. Kniesel, J. Hannemann, and T. Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd workshop on Linking aspect technology and evolution (LATE)*, page 6. ACM, 2007.

[18] R. Koschke and D. Simon. Hierarchical reflexion models. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, page 36. IEEE Computer Society, 2003.

[19] M. Lanza and S. Ducasse. CodeCrawler–an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74–94. Franco Angeli, Milano, 2005.

[20] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[21] M. A. Linton. Implementing relational views of programs. In P. B. Henderson, editor, *Software Development Environments (SDE)*, pages 132–140, 1984.

[22] M. Marin, L. Moonen, and A. van Deursen. SoQueT: Query-based documentation of crosscutting concerns. In *Proceedings of the 29th international conference on Software engineering (ICSE)*, pages 758–761. IEEE Computer Society, 2007.

[23] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM)*, pages 350–359. IEEE Computer Society Press, 2004.

[24] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views — a case study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, 2006.

[25] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis)*, pages 135–144. ACM Press, 2006.

[26] H. A. Müller and K. Klashinsky. Rigi — a system for programming-in-the-large. In *Proceedings of the 10th international conference on Software engineering (ICSE)*, pages 80–86. IEEE Computer Society Press, 1988.

[27] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[28] NDepend. Company website at: `http://www.ndepend.com`.

[29] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Trans. Softw. Eng.*, 20(6):463–475, 1994.

[30] T. W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases (ILPS)*, pages 163–196, 1993.

[31] Semmle Limited. Company website at: `http://semmle.com`.

[32] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP Views. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 275–284. IEEE Computer Society Press, 1995.