A Prolog-based Framework for Search, Integration and Empirical Analysis on Software Evolution Data

Pamela Bhattacharya Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
{pamelab,neamtiu}@cs.ucr.edu

ABSTRACT

Software projects use different repositories for storing project and evolution information such as source code, bugs and patches. An integrated system that combines these multiple repositories, along with efficient search techniques that can answer a broad range of queries regarding the project's evolution history would be beneficial to both software developers (for development and maintenance) and researchers (for empirical analyses). For example, the list of source code changes or the list of developers associated with a bug fix are frequent queries for both developers and researchers. Integrating and gathering this information is a tedious, cumbersome, error-prone process when done manually, especially for large projects. Previous approaches to this problem use frameworks that limit the user to a set of pre-defined query templates, or use query languages with limited power. In this paper, we argue the need for a framework built with recursively enumerable languages, that can answer temporal queries, and supports negation and recursion. As a first step toward such a framework, we present a Prolog-based system that we built, along with an evaluation of real-world integrated data from the Firefox project. Our system allows for elegant and concise, yet powerful queries, and can be used by developers and researchers for frequent development and empirical analysis tasks.

1. INTRODUCTION

Software projects are larger than ever and their histories run for longer than ever, so developers are overwhelmed whenever they are faced with tasks such as program understanding or searching through the evolution data for a project. Examples of such frequent development tasks include understanding the control flow, finding the list of functions that are dependent on a function, finding modules that will be affected when a module is changed during a bug-fix, etc. Similarly, during the software maintenance phase, frequent tasks include keeping track of files that are being changed due to a bug-fix, finding which developer is suitable for fixing a bug (e.g., given that he has fixed similar bugs in the past or he has worked on the modules that the bug occurs in). In addition, a framework that allows querying on integrated evolution data for large projects would be beneficial for research in empirical software engineering, where data from these repositories is frequently used for hypothesis testing. All these tasks require expressive and efficient search over large datasets that crosses repositories; therefore a framework that can integrate these data from multiple sources and answer a broad range of queries would be beneficial for both software development and empirical analysis.

While many search and visualization frameworks have been proposed, that allow efficient search and analysis on software evolution data, they have two main inconveniences: (1) they are not flexible enough, e.g., they permit a limited range of queries, or have fixed search templates; (2) they are not powerful enough, e.g., they do not allow recursive queries, or do not support negation; however, these features are essential for a wide range of search and analysis tasks. In this paper, we show how we can address these shortcomings by using a Prolog-based integration and query framework. We chose Prolog for three main reasons: (1) it is declarative, which allows elegant, concise expression of data collection and hypothesis testing, (2) it supports negation, and (3) it supports recursive queries, e.g., for computing transitive closure, which is essential in many impact analysis studies. Our framework captures a wealth of historical software evolution data (information on bugs, developers, source code), and allows concise yet broad-range queries on this data. The three main novelties of our framework are: (1) it is temporally aware; all the tuples in our database have time information that allows comparison of evolution data (e.g., how has the cyclomatic complexity of a file changed over time?); (2) it supports powerful language features such as negation, recursion, and quantification; (3) it supports efficient integration of data from multiple repositories in the presence of incomplete or missing data using several heuristics. In particular, with a single query (*Q12* in Section 4), we can now gather the bug tossing data we painstakingly collected manually in one of our prior efforts [3].

The rest of the paper is organized as follows: we discuss related work in Section 2. We describe the advantages of using a Prolog-based framework, the key novelties in our design, and our data model in Section 3. We demonstrate how our framework can elegantly express, and effectively answer, a broad range of queries, without requiring pre-defined templates, in Section 4; these queries form the kernel of a query library that can be used by developers and researchers in their activities.

We tested our framework on a large, real-world project with separate source code and bug repositories: a subset of Firefox[1] evolution data. From Firefox's source code repository we extracted change log histories to populate our source code database. We then extracted the bugs associated with these source files. Finally, we added function call edges (from the static call graph) to the database, for demonstrating how our framework is beneficial in impact analysis. In Section 5 we present preliminary results of evaluating our framework on Firefox data, in terms of result size and query speed. In Section 6 we describe open research challenges and future directions of our work.

2. RELATED WORK

Herraiz et al. [11] identified the need for organized software repositories that can improve the state-of-art data retrieval tech-

[1]Firefox (http://www.mozilla.com/firefox) is the second most widely-used web browser [6] and has been used in many empirical studies in software engineering [13, 2, 3].

niques in software engineering and ensure repeatability, traceability and third-party independent verification and validation. They proposed a research agenda by identifying the research challenges in this area.

Hindle and German [12] proposed SCQL, a first-order and temporal logic-based query language for source-code control repositories. Their data model is a directed graph that exhibits relationship between source code revisions, files and modification requests. SCQL supports universal and existential queries, as we do, but does not support negation and recursion, which we do. While we do not propose a new language, the significant difference is that we consider multiple software repositories to integrate data and answer queries. Instead of source code changes only, our framework captures relationships between three artifacts: developers, bugs and source code.

Fischer et al. [7] proposed an approach for populating a release history database that combines source code information with bug tracking data and is therefore capable of pinpointing missing data not covered by version control systems such as merge points. Similar to Fischer et al., we build our database initially by extracting information from source code and bug repositories.

German [8] proposed recovering software evolution history using software trails—information left behind by the contributors to the development process, such as mailing lists, web sites, version control logs, software releases, documentation, and the source code. The method was used to recover software evolution traits for the Ximian project. Our data collection and database population is similar to German; however our framework is a search-based tool that can answer queries aggregating data from multiple repositories.

Begel et al. [1] developed a framework named Codebook, which is capable of combining multiple software repositories within one platform. Our work is similar but the main challenge in building a framework for open source projects lies in collecting and accurately integrating related data in absence of organized repositories and missing data [7]. Their query language is restricted to regular expressions, but has support for a fixed set of pre-computed transitive closure results; we use Prolog, a Turing-complete language, hence our framework can express and answer unrestricted queries, including temporal ones.

Nussbaum et al. [14] presented the Ultimate Debian Database that integrates information about the Debian project from various sources to answer user queries related to bugs and source code using a SQL-based framework. However, their framework does not have support for queries that require negation or transitive closure.

Starke et al. [16] conducted an empirical study about programmers' search activities to identify the shortcomings of existing search tools. They found that the state-of-the art source code search tools based on the SQL-framework are not effective enough in expressing the information the developer is seeking. We believe that declarative query support will improve the code-search experience of developers.

3. FRAMEWORK

We now turn to presenting our framework. We first motivate our decision for choosing Prolog as the storage and querying engine for our framework, then describe the key novel features in our approach, followed by the data model. We implemented our framework in DES, a free, open-source Prolog-based implementation of a basic deductive database system [15].

3.1 Why Use Prolog?

Prolog is declarative. In declarative languages queries are kept concise and elegant because there is no need to specify control flow.

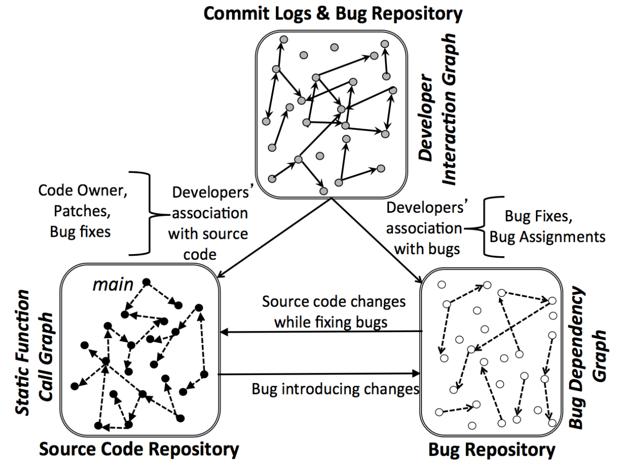

Figure 1: Information integration in our framework.

Moreover, Prolog allows flexible, broad-range queries without requiring pre-computation or having to define a set of query templates.

Prolog supports negation. Negation extends the range of expressible queries but is potentially expensive, hence existing frameworks leave it out. For example, previous frameworks cannot answer queries like "return the list of developers who have *not* fixed a bug in the past" or "return the list of modules that are *not* affected when module A is changed"; such queries are useful, however, e.g., the second query can be used to reduce regression testing. In Table 2 we show an example of negation use in our framework.

Prolog supports recursion. Recursive queries are important, e.g., for computing the transitive closure required in impact analyses. Although certain versions of SQL support recursion, it is usually a limited form of recursion, and implemented via proprietary extensions. We provide an example query that requires recursion in Table 2.

3.2 Key Features

We now showcase some key features of our framework; existing approaches fail to support one or more of these features.

3.2.1 *Temporal Queries*

Previous approaches that build databases from integrating multiple software repositories are not capable of answering temporal queries. For example, the following queries cannot be answered by existing systems: (1) who modified file A on a given day?, (2) whom was the bug B assigned to during a certain period?, (3) what changes were made to a file F during a specific period of time?, (4) how have source code metrics (e.g., complexity, defect density) of a file changed over time?

3.2.2 *Recursion*

Transitive closure is helpful for impact analysis, e.g., "return the set of files that will be affected by modifications to file F." The problem with prior approaches is that they either cannot compute transitive closure, or cannot compute it when the graph (where edges indicate a "depends" relationship) is not known statically. For example, we might want to find all the descendants of a file F after it has been refactored. If we do not know the definition of "depends", i.e., in this case, `is-descendant-of`, at the time we construct the database, we first need to write a query that generates the graph, and then transitively close it, using a language powerful enough to express transitive closure. Similarly, suppose

Table	Table Name	Attributes
Source Basic	sourcebasic	FileNameAndPath, Release, List of Functions Defined, Complexity, Defect Density, Date
Source Change	sourcechange	FileNameAndPath, Date, RevisionID, BugID, DeveloperID, Days, Lines Added
Source Depend	sourcedepend	FileNameAndPath, List of Files Depends it on (w.r.t. the static call graph), Date
Bugs	bugs	Bug ID, Date Reported, Developer ID, Date Changed, Developer Role, Severity, Bug Status, Bug Resolution, List of Dependencies, DaysReported, DaysFixed

Table 1: Database schema.

Feature	Natural Language Query	DES Clause
Negation	*Q1:* Return the list of bugs fixed by developer D which do **not** depend on other bugs	bugs_not_depend(B,D,R) :– bugs(B,_,D,_,_,_,_,_,R), **not**(R='null').
Transitive closure	*Q2:* Given two functions F1 and F2, check if a change to function F2 will affect function F1 (w.r.t. the call graph)	reach(X,Y) :– sourcedepend(X,Y). reach(X,Y) :– reach(X,Z), sourcedepend(Z,Y).

Table 2: Query support for negation and transitive closure.

we have a bug B_1 in file F, and we want to find the list of subsequent bugs in F that might have been introduced in the process of fixing B_1. The problem is, the list of subsequent bugs is constructed dynamically, e.g., all the bugs in F minus the list of bugs in F that depend on other bugs in other files. Previous approaches such as Codebook [1] use pre-computed transitive closure for efficiently answering a pre-defined set of queries, e.g., "the set of all functions F depends on"; however, queries like "list all functions that both F_1 and F_2 depends on" cannot be answered because they require language support for recursion/transitive closure.

Moreover, when data from new releases is added to the database, pre-computed transitive closure does not work, because the "depends" relationships might have changed due to the new data, hence a dynamic transitive closure algorithm would be required. Note that we are not getting expressivity for free, as transitive closure is expensive when computed naively ($O(N^3)$); however, it can be accelerated via matrix multiplication ($O(N^{\omega}log(N))$) [4], or even further with randomized algorithms [5].

3.2.3 Integration

In open source projects, it is often difficult to integrate related information because it is spatially dispersed and incomplete. For example, often bug reports do not have complete information about files that were changed during a bug fix. Consider Mozilla bug 334314; according to the Bugzilla bug report, three changes were made to file `ssltap.c` to fix this bug—once by developer ID *alexei.volkov.bugs* and twice by developer ID *nelson*. The information in the patch reference for this change is incomplete; [2] it is not clear who-has-made-which-change. However, from the change log of file `ssltap.c`, we can retrieve developers, changes, and change timestamps, which helps us complete the bug database.

As another example, consider this query: "return the list of bugs for which the developer who reported the bug fixed it." Mozilla Bug 420212 is such an example; user *Nelson Bolyard* reported the bug and developer with ID *nelson* fixed it. Entity resolution, i.e., identifying user "Nelson Bolyard" and developer "nelson" as the same person is difficult to automate. Therefore, the bug-reporting and bug-fixing tuples in our database will contain different names and bug 420212 will not be returned in the query result. To solve this problem, we use a heuristic—we check whether two names have common a substring, e.g., in this case "Nelson Bolyard" contains "nelson". However, this is not always true; for example, "matt" is a developer ID found in log files and there is more than one developer whose name or ID contain "matt", e.g., "matthewgertner" and "mattyisageek". We currently use a Perl script to find out IDs and names which are contained in each other and then manually check for false positives. In the future we plan to use automated integration heuristics (similar to Fischer et al. [7]) to increase scalability and accuracy.

3.3 Storage

As shown in Figure 1, we collect information from three sources: (1) source code repositories—size, location, source code dependencies from the static function call graph, etc., (2) bug repositories—who reported the bug, what is the present status of the bug, bug dependency data, etc., and (3) interaction between developers—who tossed bugs to whom, which two developers worked on same files, etc. Note how function calls, bugs and developer interactions induce dependency graphs. We integrate information from these three sources and store it into a database, so that our framework can answer cross-source queries, as demonstrated in Section 4. The schema for our database is presented in Table 1. We now proceed to describing the database schema, contents, and updates.

Source code. The source code data is stored in three tables: basic source code information, source code changes and source code dependencies. The *basic source code information* table (sourcebasic) stores, for each module (file): its location, the list of functions it defines, complexity metrics, defect density information, and a corresponding date. Note that a file can have multiple entries in the database due to multiple releases, hence when a file is not changed in a release, all values but the release timestamp remain unchanged. These entries are important for tracking changes between releases. In the *source change* table (sourcechange), we store details of all revisions that have been made to a file, either as feature enhancements or bug fixes: the date the change was made, the revision ID, the bug ID (if the change was due to a bug fix) and the developer who committed it, and number of lines added. For a source change entry in the database, we also store the number of days since the first commit[3] the current activity took place.[4] In the *source dependency* table (sourcedepend), we store information about which other entities a given module or function depends on directly, i.e.,

[2]Patch for bug 334314: https://bug334314.bugzilla.mozilla.org/attachment.cgi?id=218642

[3]The first commit found in the log files we used was July 23, 1998.

[4]This is done to answer queries involving time intervals.

Natural Language Query	DES Clause
Q3: Given a developer ID B, return a list of all activities (fixes F or source code changes C) associated with D	activity (B,D,F) :– sourcechange(F,D,B,_,_,_,_). activity (B,D,F) :– bugs(F,_,D,B,_,_,_,_,_,_,_).
Q4: Given a developer D, return all the bugs that he has fixed.	bugs_fixed(B,D,R) :– bugs(B,_,D,'Fixed',_,_,_,_,_,_,_).
Q5: Return the list of bugs developer D has been assigned on date DT	bugs_fixed_bydate(B,D,DT) :– bugs(B,_,D,'Assigned',_,DT,_,_,_,_,_).
Q6: Return the list of bugs developer D could **not** fix	bugs_not_fixed(B,D) :– bugs(B,_,D,'Assigned',_,_,_,_,_,_,_).
Q7: Return the list of bugs developer D reported and was eventually fixed by E	bugs_fixed_D_E(B,D,E) :– bugs(B,_,D,'Reported',_,_,_,_,_,_,_), bugs(B,_,E,'Fixed',_,_,_,_,_,_,_).
Q8: Return the list of files modified by developer D on date DT	source_modified_bydate(F,D,R,DT) :– sourcechange(F,D,_,R,DT,_,_).
Q9: Return the list of files modified by developer D for which more than 10 lines were added	source_modified_bylines(F,D,B,R,L) :– sourcechange(F,D,B,R,_,_,L), L>10.
Q10: Return all source file changes	all_src_changes(B,R,F,DT,D) :– sourcechange(F,D,B,R,DT,_,_).
Q11: Return the list of bugs reported and fixed by the same developer D	bugs_fixed_D_D(B,D) :– bugs(B,_,D,'Reported',_,_,_,_,_,_,_), bugs(B,_,D,'Fixed',_,_,_,_,_,_,_).
Q12: Return the tossing history of bug B	bugs_toss(B,D,R) :– bugs(B,_,D,R,_,_,_,_,_,_,_).
Q13: Return the source files that have been modified by two developers D and E	common_modified(D,E,R) :– sourcechange(R,D,_,_,_,_,_), sourcechange(R,E,_,_,_,_,_).
Q14: Return the list of bugs reported between two dates D1 and D2	bugs_reported_bydate(B,D,DT) :– bugs(B,_,D,'Reported',_,_,_,_,_,DT,_), DT<D2, DT>D1.
Q15: Return the list of bugs fixed between dates D1 and D2	bugs_fixed_bydate(B,D,DT) :– bugs(B,_,D,'Fixed',_,_,_,_,_,DT,_), DT<D2, DT>D1.
Q16: Return the list of source files modified by developer D before date D1	source_modified_bydate(F,D,R,DT,DY) :– sourcechange(F,D,_,R,DT,DY,_), DY<D1, DY>0.
Q17: Return the list of open (unresolved) bugs in the database	bugs_new(B,D) :– bugs(B,_,D,_,_,_,_,_,_,_,−1).

Table 3: Sample queries from our library.

file, module or function dependencies induced by the call graph.

Bugs. The bug table (bugs in Table 1) stores information related to a bug: the date on which the bug was reported, list of developers associated with the bug and their roles (i.e., who reported it, who the bug was assigned to at some point, who fixed it), the severity of the bug, the present status of the bug, final resolution of bug and list of bugs this bug depends on. To answer queries about a time interval (e.g., how many bugs were fixed between July 2008 and May 2010), we add two attributes —DaysReported and DaysFixed—that represent the number of days since the first release of the project that the bug was reported and fixed respectively. If a bug has not been resolved at the time of database creation, DaysFixed is set to -1.

Developer information. Although Figure 1 shows data dependencies among three repositories, due to our source and bug table schema design choice, having a developer database is redundant. All the information for developers (e.g., tossing information, bug fix information, code authorship information) can be extracted from the source code and bug tables.

Updating the database. As software evolves, our database needs to grow; note that the database is monotonically increasing (we never retract facts).

4. EXAMPLES

We now proceed to presenting use cases for our system—a variety of frequent queries that arise in software development and empirical research. In Table 3 we demonstrate how using Prolog improves expressiveness and allows arbitrary information retrieval, without the need for pre-computation or templates. For example, query *Q4* returns a list of bugs that developer D had been assigned and could successfully fix. Similarly, *Q5* computes the list of bugs that developer D could not fix and were eventually fixed by developer E. We envision these queries forming the kernel of a query library that can be used by developers in their daily development and maintenance activities; similarly, the library can be useful to researchers for empirical analysis and hypothesis testing. While Table 2 has showcased negation and recursion, Table 3 showcases quantification: since our query language is based on Prolog, we support existential queries directly (variables in Prolog clause heads are existentially quantified), and universal queries by rewriting, i.e., $\forall x\, Q(x) \Leftrightarrow \neg \exists x\, \neg Q(x)$.

5. RESULTS

We randomly selected 2128 C files and 58 C++ files from the Firefox source code repository and extracted their complete change log histories to populate our source change database. We extracted the bugs associated with these source files, resulting into 932 bug files for our bug database. We also added to our source dependency table the 50 function call edges induced by the static call graph between functions in these files. In total, our database contained

Query		Resulting tuples	Time (ms)
Q1	bugs_not_depend(B,wtc,R)	218	1,746
Q2	reach('main;nsinstall .c', 'PK11_FreeSlot;pk11slot.c')	1	4
	reach('PK11_FreeSlot;pk11slot.c', 'main;nsinstall .c')	0	5
Q3	activity (B,wtc,F)	2,569	4,489
Q4	bugs_fixed(B,wtc)	218	143
Q5	bugs_assigned_bydate(B,wtc, '2006/02/02')	1	109
Q6	bugs_not_fixed(B,wtc)	558	287
Q7	bugs_fixed_D_E(B,fabientassin,wtc)	1	127
Q8	source_modified_bydate(F,nelson, R,'2001/01/07')	46	197
Q9	source_modified_bylines(F,wtc,B,R,L)	1,312	695
Q10	all_src_changes(B,R,F,DT,D)	39,866	733,566
Q11	bugs_fixed_D_D(B,nelson)	126	25
Q12	bugs_toss(236613,D,R)	18	143
Q13	common_modified(nelson,wtc,R)	465	25,120
Q14	bugs_reported_bydate(B,D,DT), 2008/7/23<DT< 2008/10/23 .	50	875
Q15	bugs_fixed_bydate(B,D,DT), 2008/7/23<DT< 2008/10/23 .	47	1,769
Q16	source_modified_bydate(F,nelson, R,DT,DY), DT=2008/7/23.	1,275	1,282
Q17	bugs_new(B,D)	810	2,435

Table 4: Example queries for query declarations in Table 3.

63,142 tuples. In Table 4 we present the queries we used to test the query definitions showed in Table 3. The first column shows the query invocation, the second column shows the number of resulting tuples, [5] and the third column shows the query execution time, in milliseconds.

We found that the time taken to answer a query using DES increases with the increase in number of resulting tuples, hence it can be quite high for queries with large results, e.g., *Q10*; we plan to address this scalability issue in future work.

6. FUTURE WORK

We are currently using DES, an open-source Prolog-based implementation of deductive databases [15] as our framework's engine. In the future, we plan to use the bdddbddb framework to speed up DES queries [17], as bdddbddb has been shown to be able to handle Datalog-based static analyses for large, real-world programs. We plan to use other software traits/trails, e.g., mailing list information, to improve our data set for more accurate information modeling and retrieval. As discussed in Section 3.2.3, similar to Fischer et al. [7], we plan to design heuristics that can pinpoint missing data and combine non-explicit co-evolved data from various sources (as demonstrated with an example in Section 3.2.3). In our preliminary experiments as shown in Section 5, we did not use the sourcebasic database or any queries related to it. In future, we would like to extend our library to answer queries related to the sourcebasic like: "which file exhibited the maximum increase in complexity or defect density during a given time interval." Additionally, as shown in Figure 1, we would also like to track bug-introducing changes using our framework—changes in the source code that led to bugs. We would like to include code-ownership information to indicate which developer owns which artifact of a software system in our database using heuristics similar to Girba et al. [9]. Finally, we plan to add a visualization layer [10] on top of our current framework that will allow query results to be displayed visually, rather than as text.

[5] In query *Q2* in Table 4, $Func; Mod$ represents function $Func$ defined in module Mod; the resulting tuple 1 denotes there is a path from $F_1; M_1$ to $F_2; M_2$ while 0 denotes otherwise.

7. CONCLUSION

In this paper we show how using a Prolog-based framework we can answer a broad range of queries on software evolution data that cross multiple software repositories. We used several examples on Firefox source and bug repositories to show how our framework is efficient in querying large, real-world evolution data. In the future, we would like to improve the scalability of our framework, increase its precision, and add a visualization component.

8. REFERENCES

[1] A. Begel, K. Y. Phang, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *ICSE*, 2010.

[2] P. Bhattacharya and I. Neamtiu. Assessing programming language impact on development and maintenance: A study on C and C++. In *ICSE 2011*.

[3] P. Bhattacharya and I. Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *ICSM*, 2010.

[4] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. 2nd edition, 2001.

[5] C. Demetrescu and G. Italiano. Fully dynamic transitive closure: breaking through the o(n2) barrier. In *FOCS'00*.

[6] Firefox Statistics. http://www.computerworld.com/s/article/9140819/1_in_4_now_use_Firefox_to_surf_the_Web.

[7] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM*, 2003.

[8] D. M. German. Using software trails to reconstruct the evolution of software. *JSME'04*.

[9] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *IWPSE*, 2005.

[10] M. Goeminne and T. Mens. A framework for analysing and visualising open source software ecosystems. In *EVOL/IWPSE*, 2010.

[11] I. Herraiz, G. Robles, and J. M. Gonzalez-Barahona. Research friendly software repositories. In *IWPSE-Evol*, 2009.

[12] A. Hindle and D. M. German. SCQL: a formal model and a query language for source control repositories. In *MSR'05*.

[13] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3), 2002.

[14] L. Nussbaum and S. Zacchiroli. The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In *MSR*, 2010.

[15] F. Saenz-Perez. DES: A Deductive Database System. In *PROLE*, 2010.

[16] J. Starke, C. Luce, and J. Sillito. Working with search results. In *SUITE*, 2009.

[17] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.