Evaluating Recommended Applications

Mark Grechanik
System Integration Group
Accenture Technology Labs
Chicago, IL 60657
mark.grechanik@accenture.com

Denys Poshyvanyk
Department of Computer Science
The College of William and Mary
Williamsburg, VA, 23185
dposhyvanyk@wm.edu

ABSTRACT
Large open source software repositories are polluted with incomplete or inadequately functioning projects having scarce or poor descriptions. Developers often search these repositories to find sample applications containing implementations of relevant features. While relying on software search engines that retrieve germane applications based on direct matches between user queries and words in the descriptions (or source code files), it is difficult to warrant that retrieved applications contain functionality described by their authors in project summaries.

We propose a novel approach called K9 that helps users evaluate if recommended applications implement functionality, which is described by the authors in project summaries. Using programmers' queries describing high-level concepts (e.g., `sql server`, `floating toolbar`, or `smart card`), K9 uses information retrieval and program analysis techniques to evaluate recommended applications based on how they implement these high-level concepts. We conjecture that K9 will effectively support developers in evaluating functionality of the retrieved applications.

Categories and Subject Descriptors
D.2.13 [**Software Engineering**]: Reusable Software – *reusable libraries.*

General Terms
Measurement, Documentation

Keywords
Open Source Software Repositories, Program Analysis, Information Retrieval, Application Programming Interface, Software Documentation

1. INTRODUCTION
When depositing applications into software repositories, programmers create meaningful descriptions of these applications. A fundamental problem of finding relevant applications is in the mismatch between the high-level intent reflected in the descriptions of these applications and low-level implementation details. Many application repositories are polluted with poorly functioning projects [7], and matches between keywords from the queries with words in the descriptions of the applications do not guarantee that these applications behave as their authors claim.

Currently, the only way for programmers to determine if an application behaves as its authors claim is to download this application, locate and examine fragments of its code that implement the desired features and/or observe the runtime behavior of this application to ensure that this behavior matches requirements. This process is usually manual with programmers studying the source code of the retrieved applications, locating various *Application Programming Interface (API)* calls, and reading information about these calls in help documents. Still, it is difficult for programmers to link high-level concepts from requirements to their implementations in source code [1].

Modern search engines do little to ensure that retrieved applications behave as their authors claim in their descriptions. To assist programmers with evaluation of applications recommended by search engines, code mining systems should ensure that recommended applications have functionality that is described by these requirements, and they should help programmers verify this functionality.

K9 is a system that helps users evaluating recommended applications that behave as their authors describe in project summaries. K9 combines information retrieval and program analysis techniques to link high-level concepts specified in the user queries to relevant APIs in help documents and their usage in the source code of these applications. We utilize this information to rank applications retrieved by source code search engines and supply programmers with source code fragments implementing these concepts. We are currently building the infrastructure for the K9 system, and are planning on evaluating K9 through user studies involving specific software reuse tasks.

2. Our Approach
In this section we provide the key ideas behind K9 and summarize its architecture.

2.1 Key Ideas
Our goal is to automate parts of the human-driven procedure of evaluating recommended applications. Suppose that requirements specify that a program should encrypt and compress data. When retrieving applications from repositories, such as Sourceforge[1]

RSSE'08, November 10, Atlanta, Georgia, USA

[1] http://sourceforge.net

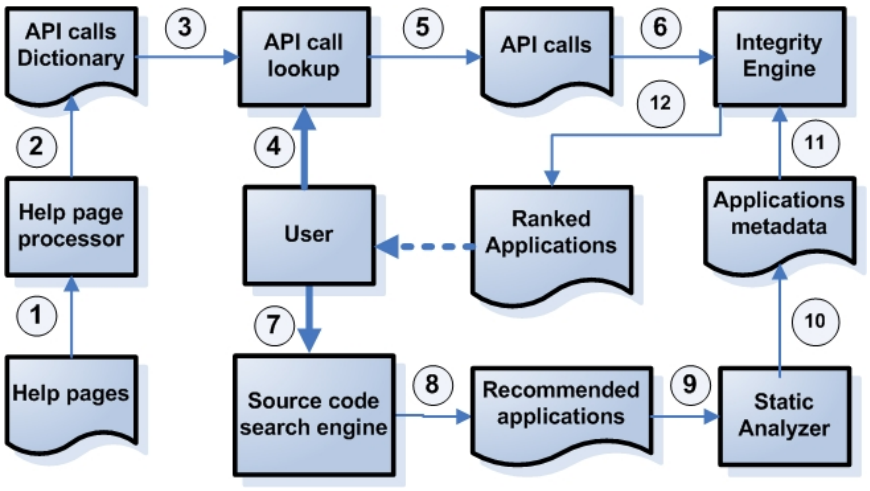

Figure 1. Overview of K9 architecture

using the keywords *encrypt* and *compress*, programmers look into the source code to check if some API calls (e.g., APIs from third-party packages) are used to encrypt and compress data. Even though the presence of these API calls does not guarantee that the applications behave as their authors claim, it is a good starting point for deciding whether to inspect these applications further.

Our idea is to use help documents that describe API calls to match keywords from user queries to the words in the descriptions of these API calls. Help documentation is usually supplied by the same vendors whose packages are used in a multitude of software applications. When programmers read these help documents about API calls, they trust these documents since they come from known and respected vendors, and are written or reviewed by multiple people and used by other programmers who report their experience at different forums. Because of these and some other factors, help documents are more verbose and accurate, and are consequently trusted more than the descriptions of applications from software repositories.

We observe that relations between concepts present in user queries are often preserved as dataflow links between API calls that implement these concepts in the program code. The main idea behind K9 is to determine relations (i.e., dataflow links) between API calls in retrieved applications. If a dataflow link is present between two API calls in the program code of one application and there is no link between the same API calls in some other application, then the former application should have a higher ranking than the latter. We hypothesize that it is possible to achieve a higher precision in evaluating recommended applications by using this heuristic to rank applications, and we plan to evaluate this claim in the future case studies.

2.2 K9 Architecture and Process

Our approach consists of five major components: help page processor, API lookup engine, source code search engine, static analyzer and integrity engine. An overview of all the components is presented in Figure 1.

The initial step while using K9 system is in indexing databases of help documents with *help page processor*. Help page processor is a crawler that indexes help documents (1) that come from the Java API documentation[2], MSDN library[3] as well as documentation from other third-party vendors. The output (2) of the help page processor is a dictionary of API calls, which is represented by a set of tuples (($word_1$, . . ., $word_n$), API call) linking API calls with their authentic descriptions (i.e., set of words) extracted from help documents. Once the API calls dictionary is constructed (or updated), the K9 system is ready to accept queries from users. Our approach for mapping words to API calls is different from the *keyword programming* technique presented in [8] as we derive mappings between words and APIs from external documentation rather than source code.

When the user issues a query (4) into K9, it is passed to *API call lookup* and *Source code search* engines. Subsequently, the lookup engine scans the API calls dictionary using the words from the query as keys and outputs the set of API calls, which contains words in the descriptions matching the words in the original user query (5). On the other hand, the same query is passed to the *Source code search* engine (e.g., Google Code Search[4] or Koders[5]) to retrieve a set of initial software applications. Afterward, a *Static Analyzer* executes the API dataflow link heuristic on all of the retrieved software systems to generate applications metadata. The application metadata contains dataflow links between different API calls, which appear in the source code of recommended applications.

Both the API calls from the step (6) and applications metadata from the step (11) are supplied into the *Integrity Engine* as an input. The engine uses a set of heuristics to compare the API calls that are relevant to user queries with API calls, which appear in the source. The engine ranks all the recommended applications based on the frequencies of relevant API usages as well as API connectivity data flow measures. The idea behind this ranking

[2] http://java.sun.com/j2se/1.4.2/docs/api/
[3] http://msdn.microsoft.com/en-us/library/
[4] http://www.google.com/codesearch
[5] http://www.koders.com/

mechanism is that the software applications using APIs, which are relevant to user queries, are ranked higher.

3. RELATED WORK

Several different code mining techniques and tools have been proposed to retrieve relevant components or applications from software repositories. The number of tools has been developed to retrieve code snippets based on the user queries (e.g., PARSEWeb [15]) or the context of the source code that programmers work on (e.g., Prospector [9], Hippikat [3], XSnippet [12], Gilligan [6], Jigsaw [2] and Strathcona [5]). The tools, such as CodeFinder [4] and CodeBroker [16] retrieve relevant software components based on their descriptions and source code respectively. Other tools, such as Mica [14] also use external search engines (e.g., Google) for finding relevant applications. A set of existing approaches to feature location concentrate on identifying source code snippets, which implement particular high-level domain concepts, in a given software system [10, 11, 13, 17]. The K9 is different from the above mentioned tools and approaches as it operates on application level granularity (i.e., analyzes and retrieves software projects) and also attempts evaluating functionality of the retrieved projects.

4. CONCLUSIONS AND FUTURE WORK

In this position paper we proposed a novel approach called K9 for evaluating functionality of retrieved applications. K9 combines techniques from program analysis and information retrieval to rank higher applications that implement high-level concepts entered as a part of the user queries. We are planning on evaluating K9 in a set of case studies with specific software reuse tasks.

5. ACKNOWLEDGMENTS

This research was supported in part by the United States Air Force Office of Scientific Research under grant number FA9550-07-1-0030. The authors would like to thank to the anonymous reviewers for pertinent and helpful comments and suggestions.

6. REFERENCES

[1] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E. The Concept Assignment Problem in Program Understanding in Proceedings of 15th IEEE/ACM International Conference on Software Engineering (ICSE'94) (May 17-21, 1994), 482-498.

[2] Cottrell, R., Walker, R., and Denzinger, J. Jigsaw: A tool for small-scale source code reuse in Proceedings of 30th IEEE/ACM International Conference on Software Engineering (ICSE'08) (2008), 933-934.

[3] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S. Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering*, 31, 6 (June 2005), 446-465.

[4] Henninger, S. Supporting the construction and evolution of component repositories in Proceedings of 18th IEEE/ACM International Conference on Software Engineering (ICSE'96) (1996), 279 - 288.

[5] Holmes, R. and Murphy, G. C. Using structural context to recommend source code examples in Proceedings of 27th International Conference on Software Engineering (ICSE'05) (2005), 117- 125.

[6] Holmes, R. and Walker, R. Supporting the investigation and planning of pragmatic reuse tasks in Proceedings of 29th IEEE/ACM International Conference on Software Engineering (ICSE'07) (2007), 447-457.

[7] Howison, J. and Crowston, K. The perils and pitfalls of mining SourceForge in Proceedings of 1st International Workshop on Mining Software Repositories (MSR'04) (2004).

[8] Little, G. and Miller, R. C. Keyword programming in java in Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07) (Atlanta, GA, 2007), 84-93.

[9] Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. Jungloid mining: helping to navigate the API jungle in Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05) (2005), 48 - 61.

[10] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V. Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering*, 33, 6 (June 2007), 420-432.

[11] Robillard, M. Automatic Generation of Suggestions for Program Investigation in Proceedings of Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (Lisbon, Portugal, September, 2005), 11 - 20

[12] Sahavechaphan, N. and Claypool, K. XSnippet: mining For sample code in Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06) (2006), 413 - 430.

[13] Shepherd, D., Fry, Z., Gibson, E., Pollock, L., and Vijay-Shanker, K. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns in Proceedings of International Conference on Aspect Oriented Software Development (AOSD'07) (2007), 212-224.

[14] Stylos, J. and Myers, B. A. Mica: A Web-Search Tool for Finding API Components and Examples in Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (2006), 195- 202.

[15] Thummalapenta, S. and Xie, T. Parseweb: a Programmer Assistant for Reusing Open Source Code on the Web in Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07) (Atlanta, GA, 2007), 204-213.

[16] Ye, Y. and Fischer, G. Reuse-Conducive Development Environments. *Journal Automated Software Engineering*, 12, 2 (2005), 199-235.

[17] Zhao, W., Zhang, L., Liu, Y., Sun, J., and Yang, F. SNIAFL: Towards a Static Non-interactive Approach to Feature Location. *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, 15, 2 (2006), 195-226.