Creating and Evolving Software by Searching, Selecting and Synthesizing Relevant Source Code

Denys Poshyvanyk[1], Mark Grechanik[2, 3]

[1]Computer Science Department
The College of William and Mary
Williamsburg, VA 23185
denys@cs.wm.edu

[2]Accenture Technology Labs
Chicago, IL 60657
mark.grechanik@accenture.com

[3]Computer Science Department
University of Illinois, Chicago,
drmark@uic.edu

Abstract

When programmers develop or maintain software, they instinctively sense that there are fragments of code that other developers implemented somewhere, and these code fragments could be reused if found.

*In this paper, we propose a novel solution that addresses the fundamental questions of **s**earching, **s**electing, and **s**ynthesizing (S^3) software based on the analysis of Application Programming Interface (API) calls as units of abstractions that implement high-level concepts (e.g., the API call EncryptData implements a cryptographic concept). This paper outlines the details behind S^3, analyzes current challenges and describes evaluation plans.*

1. Introduction

Creating software from existing components rather than building it from scratch is a fundamental problem of software reuse. Currently, the source code of hundreds of thousands of applications is publicly available to programmers for reuse. It is estimated that around one trillion lines of code have been written to date with 35 billion lines of source code being written every year (see Grady Booch's keynote speech at AOSD'05 on "The Complexity of Programming Models"). Naturally, when programmers develop software, they instinctively sense that there are fragments of code that other programmers wrote, and these fragments can be reused.

The there main problems that inhibit effective mainstream software reuse practices are how to search source code effectively, how to select retrieved code snippets from relevant retrieved applications, and how to bridge the abstraction gap between design and low level implementations. Moreover, source code repositories are polluted with poorly functioning projects with incomplete descriptions or documentation, if present at all. State-of-the-art code search engines (e.g., Google Code Search) match words from search queries to the identifiers or words in comments in open-source projects. Unfortunately, these engines provide no guarantee that found code snippets implement concepts or features described in queries.

Even if relevant source code fragments are located precisely in billions of lines of existing open-source code, developers face another daunting task of moving these fragments into their software by hand as these code fragments may exhibit completely different behavior in the contexts of different applications. In addition, synthesizing new code by composing selected code fragments with each other requires sophisticated reasoning about the behavior of the fragments and the resulting code. The result of this process is overwhelming complexity, steep learning curve, and the significant cost of building customized software.

We propose a novel approach that addresses the fundamental questions of **s**earching, selecting, and **s**ynthesizing software based on a common abstraction and behavior-specific compositional mechanisms. This approach is based on the fact that programs heavily use well-known third-part API calls to implement high-level requirements. These API calls represent units of abstractions, and these abstractions describe common requirements (e.g., encrypting and sending XML data over the network). Using these abstractions unifies *searching* (S_1), *selecting* (S_2), and *synthesizing* (S_3) applications in a novel and promising way: *searching* returns applications that contain API calls that implement requirements specified in the search query, *selecting* code fragments is centered on these located API calls and dataflow dependencies among them, and code *synthesis* exploits static program analysis and runtime information to guide programmers in composing code fragments effectively.

2. Searching, selecting, and synthesizing

Using APIs has become a large part of everyday programming for millions of software developers [24]. The number of API calls that are exposed by different software packages is measured in hundreds of thousands. For instance, Microsoft Windows and Java Development Kits have collectively over 50,000 API calls, and their number is growing on a daily basis. Retrieving, indexing, and analyzing information about

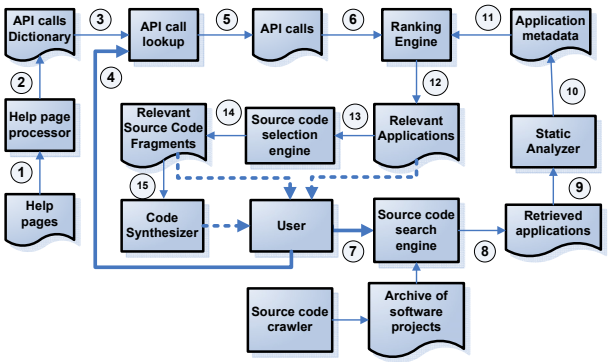

Figure 1. Overview of the S[3] architecture

API calls is necessary to support developers who create and maintain large software systems since these systems utilize various APIs. In order to comprehensively address the challenges of searching, selecting, and synthesizing code, an approach should rely on the information which is derived from analysis of API calls described in the software documentation.

We observe that relations between concepts that are entered as keywords in queries are often preserved as dataflow links between API calls that implement these concepts in source code. This observation is closely related to the concept of *software reflection models*, formulated by Murphy, Notkin, and Sullivan, where relations between elements of high-level models are preserved in their implementations in source code [17]. Our idea of improving the relevance of search results 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 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 finding relevant applications by using this heuristic to rank applications, and we are planning on thoroughly evaluating our hypothesis.

The initial step while using the S_1 (i.e., *searching*) component of S^3 is in indexing databases of help documents with a *help page processor*. A help page processor is a crawler that indexes help documents (1) that come from the Java API documentation[1], MSDN library[2] 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 descriptions (i.e., set of words) that are extracted from help documents. Once the dictionary of APIs is constructed (or updated), the system can accept queries from users. Our approach for mapping words to API calls is different from the *keyword programming* technique [14], since we derive mappings between words and APIs from external documentation rather than source code.

When a user issues a query (4) into S_1, 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 that match the words in the original user query (5).

To accomplish searching, we use two *Information Retrieval (IR)* methods: Latent Semantic Indexing [5] and index-based retrieval (e.g., Apache Lucene[3]). In addition, we use the Google Code search[4] to retrieve a set of initial software applications (7). We are also building and testing our version of a source code crawler for downloading, extracting and indexing open-source applications from open-source repositories, such as sourceforge[5] [7]. The progress on downloading and indexing open-source software is presented in Table 1.

Once the database is populated with projects from open-source repositories, we will run a set of case studies for evaluating and fine-tuning different heuristics for retrieving relevant applications.

The next step in using S_1 is to execute the API dataflow link heuristics with a *Static Analyzer* on retrieved applications to generate applications metadata. The application metadata contains dataflow links between different API calls, which appear in the source code of retrieved applications.

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

Table 1. Statistics on downloading and indexing open-source projects from Sourceforge.net as of 11/26/08

Items	Count
Java projects	21,934
Files	38,330
Files downloaded (*.zip, *.tar.gz, etc)	31,371
Files skipped (*.exe, *.dmg, *.pdf, etc)	6,959
GB downloaded	105.62Gb
GB skipped	45.71Gb
Files indexed in Lucene	10,897
Java docs in index	100,866

[1] http://java.sun.com/j2se/1.4.2/docs/api/
[2] http://msdn.microsoft.com/en-us/library/
[3] http://lucene.apache.org/
[4] http://www.google.com/codesearch
[5] http://sourceforge.net/

behind this ranking mechanism is that the software applications that use APIs which are relevant to user queries are ranked higher.

Once the list of candidate relevant applications is obtained, users inspect them and select code fragments that are relevant to the initial queries. Our idea behind the S_2 (i.e., *selecting*) component of S^3 is to use data that is extracted using textual, static, and dynamic analyses (e.g., using an existing feature location technique [18]) as well as additional information on connectivity and distributions of API calls, which is retrieved using S_1, to identify relevant code fragments in source code. We will investigate the complementary roles for the different sources of information used in the implementation of S_2: textual, dynamic as well as information on relevant API calls detected in retrieved software systems.

Once a code fragment is selected and extracted using S_2, it will be saved as a function with input and output parameters and synthesized using S_3 component. Currently, we are exploring some of the existing solutions for traceability link recovery, specifically the *LeanArt* approach [8], in the context of code synthesis, which is based on a combination of program analysis, run-time monitoring, and machine learning techniques.

3. Evaluation Plans

Further research activities include rigorous empirical validation of the proposed S^3 approach and its accompanying techniques. Among several available empirical techniques, case studies are predominantly suitable for the validation of the proposed research. Case studies are used superlatively in contexts where there is little control over variables [29]. We are planning a set of exploratory and descriptive case studies aimed at building, explaining, and validating the proposed technique. The following research questions pertinent to S^3 will be studied: (1) how does S^3 improve user searches for relevant applications, and through them how does it impact software reuse?; (2) which ranking heuristics are best suited for retrieving relevant applications using S_1?; (3) how to present relevant code fragments to software developers using S_2 and how to verify that behavior of the selected code fragments is correct?; (4) how to overcome a cognitive distance for selecting and synthesizing code fragments using S_2 and S_3?; (5) how to assist software developers in synthesizing selected fragments S_3 into a working copy of a software system? The case study designs will contain research questions, study propositions, units of analysis logic of linking the data to the propositions, and criteria for interpreting the findings [29].

4. Related Work

In this section, we summarize related work to each part of the S^3 approach: approaches that search source code for reuse, approaches that locate and select fragments of relevant source code, and some of the related work on code synthesis.

Different code mining techniques and tools have been proposed to retrieve relevant software components from different repositories. Some of these tools are CodeFinder [9], CodeBroker [28], Mica [25], Prospector [15], Hipikat [4], xSnippet [21], Starthcona [12], AMC [11], SPARS-J [13], Google code search, Sourcerer [2], Exemplar [7] and ParseWeb [26]. These tools can be broadly classified by the granularity of the search: fragments of source code [11, 12, 15, 21, 25, 26], modules [9, 28], applications [2, 4]; scope of the search: source code [2, 11, 12, 15, 25, 26], documentation [9, 21, 28] or both [4]; granularity of input queries: APIs [11, 12, 15, 21, 26] or natural language keywords [9] [2, 4, 25, 26, 28]. The S_1 component is different from these existing search tools as it allows searchers to use both granularities (fragments and applications), flexible user queries consisting of API calls and keywords, and it utilizes not only source code but also its documentation.

Existing approaches to concept location, which are pertinent to the S_2 component, can be broadly classified into three categories based on the type of information that they use: static [16] [22] [3] [20] [19], dynamic [1, 27] and hybrid [6] [18] [10, 30] methods which combine static and dynamic analyses. Selecting pertinent code fragments (or complete features) from retrieved applications is a research goal behind the S_2 component. While existing feature location techniques mainly aim at identifying a small number of feature components (e.g., methods) in a single software project, the proposed research on S_2 aims at locating relevant code fragments in a *set of retrieved applications.*

While several existing solutions to code synthesis have been proposed in the literature [15] [23] that are directly related to S_3 component of the model, our solution to synthesizing selected code fragments will be based on the existing solution combining program analysis, run-time monitoring, and machine learning, implemented in the *LeanArt* approach [8].

5. Conclusions and Future Work

This paper proposes a novel approach, namely S^3, that unifies searching, selecting, and synthesizing applications in a powerful and novel way: searching returns applications that contain API calls that implement requirements specified in a search query, selecting code fragments is centered around found API

calls and dependencies (textual, structural, and dynamic) among them, and code synthesis exploits static program analysis, runtime information and machine learning to guide programmers in composing these code fragments more effectively. This paper outlines some of the plans for evaluating the proposed S^3 technique together with existing challenges for implementing different components of the model.

6. Acknowledgements

This research was supported in part by the United States Air Force Office of Scientific Research under grant number FA9550-07-1-0030.

7. References

[1] Antoniol, G. and Guéhéneuc, Y. G., "Feature Identification: An Epidemiological Metaphor", *IEEE Transactions on Software Engineering*, vol. 32, no. 9, 2006, pp. 627-641.

[2] Baldi, P., Linstead, E., Lopes, C., and Bajracharya, S., "A Theory of Aspects as Latent Topics", in Proc. of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, Nashville, TN, 2008, pp. 543-562.

[3] Chen, K. and Rajlich, V., "Case Study of Feature Location Using Dependence Graph", in Proc. of 8th IEEE International Workshop on Program Comprehension, Limerick, June 2000, pp. 241-249.

[4] Cubranic, D., Murphy, G. C., Singer, J., and Booth, K. S., "Hipikat: A Project Memory for Software Development", *IEEE Transactions on Software Engineering*, vol. 31, no. 6, June 2005, pp. 446-465.

[5] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[6] Eisenbarth, T., Koschke, R., and Simon, D., "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, vol. 29, no. 3, March 2003, pp. 210 - 224.

[7] Grechanik, M., Conroy, K. M., and Probst, K. A., "Finding Relevant Applications for Prototyping", in Proc. of 4th IEEE International Workshop on Mining Software Repositories (MSR'07), Minneapolis, MN, 2007, pp. 12-15.

[8] Grechanik, M., McKinley, K. S., and Perry, D., "Recovering and using use-case-diagram-to-source-code traceability links", in Proc. of 6th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07), 2007, pp. 95-104.

[9] Henninger, S., "Supporting the construction and evolution of component repositories", in Proc. of 18th IEEE/ACM International Conference on Software Engineering, 1996, pp. 279 - 288.

[10] Hill, E., Pollock, L., and Vijay-Shanker, K., "Exploring the Neighborhood with Dora to Expedite Software Maintenance", in Proc. of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), November 2007, pp. 14-23.

[11] Hill, R. and Rideout, J., "Automatic Method Completion", in Proc. of 19th International Conference on Automated Software Engineering (ASE'04), September 20-24 2004, pp. 228- 235.

[12] Holmes, R., Walker, R. J., and Murphy, G. C., "Approximate Structural Context Matching: An Approach to Recommend Relevant Examples", *IEEE Transactions on Software Engineering*, vol. 32, no. 12, Dec. 2006, pp. 952-970.

[13] Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M., and Kusumoto, S., "Ranking significance of software components based on use relations", *IEEE Transactions on Software Engineering (TSE)*, vol. 31, no. 3, March 2005, pp. 213- 225.

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

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

[16] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in Proc. of 11th IEEE Working Conference on Reverse Engineering, Delft, Netherlands, Nov. 9-12 2004, pp. 214-223.

[17] Murphy, G. C., Notkin, D., and Sullivan, K. J., "Software Reflexion Models: Bridging the Gap between Design and Implementation", *IEEE Transactions on Software Engineering* vol. 27, no. 4, 2001, pp. 364-380.

[18] 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*, vol. 33, no. 6, June 2007, pp. 420-432.

[19] Robillard, M. P., "Topology Analysis of Software Dependencies", *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 4, August 2008.

[20] Robillard, M. P. and Murphy, G. C., "Concern Graphs: Finding and describing concerns using structural program dependencies", in Proc. of IEEE/ACM ICSE'02, pp. 406–416.

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

[22] 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 Proc. of International Conference on Aspect Oriented Software Development (AOSD'07), 2007, pp. 212-224.

[23] Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., and Seshia, S., "Sketching stencils", in Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, California, USA, 2007, pp. 167-178.

[24] Stylos, J. and Myers, B., "The Implications of Method Placement on API Learnability", in Proc. of 16th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'08), Atlanta, GA, November 9-14 2008.

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

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

[27] Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D., "Locating User Functionality in Old Code", in Proc. of IEEE International Conference on Software Maintenance (ICSM'92), Orlando, FL, November 1992, pp. 200-205.

[28] Ye, Y. and Fischer, G., "Reuse-Conducive Development Environments", *Journal Automated Software Engineering*, vol. 12, no. 2, 2005, pp. 199-235.

[29] Yin, R. K., *Applications of Case Study Research*, 2 ed ed., CA, USA, Sage Publications, Inc, 2003.

[30] 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)*, vol. 15, no. 2, 2006, pp. 195-226.