Exploiting Code Search Engines to Improve Programmer Productivity

Suresh Thummalapenta

Department of Computer Science
North Carolina State University
Raleigh, USA
sthumma@ncsu.edu

Abstract

Code Search Engines (CSE) can serve as powerful resources of open source code, as they can search in billions of lines of open source code available on the web. The strength of CSEs can be used for several tasks like searching relevant code samples, identifying hotspots, and finding bugs. However, the major limitations in using CSEs for these tasks are that the returned samples are too many and they are often partial. Our framework addresses the preceding limitations and thereby helps in using CSEs for these tasks. We showed the effectiveness of our framework with two tools developed based on our framework.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Object-oriented programming **General Terms:** Languages, Experimentation.
Keywords: Code reuse, Code search engine, Code examples, hotspots

1. Introduction

Various Code Search Engines (CSEs)[1] are available on the web that can search in billions of lines of available open source code. Given a query, CSEs can suggest relevant code samples with usages of keywords in the given query. But these CSEs are not quite helpful in practice, as they often produce a large number of code examples for a given query and the desired code sample is often not available among the first several results. For example, the Google Code Search Engine (GCSE)[2] returns near 3000 samples for the query `lang:java java.sql.Statement executeUpdate`.

Along with assisting programmers in reusing code samples from existing frameworks or libraries, the code samples returned by CSEs can also be used for other tasks like identifying framework hotspots and finding bugs in projects that reuse open source frameworks or libraries. Exploiting CSEs for these tasks requires analysis of code samples returned by CSEs. This analysis of code samples is non-trivial because the code samples returned by CSEs are often partial. The reason for partial code samples is that CSEs retrieve only source files with usages of the given query instead of entire projects. Due to the preceding limitations, traditional code analysis cannot be applied for analyzing the gathered code samples. Therefore, our framework uses several heuristics (Section 2) for analyzing these code samples and transforms them into an intermediate form. We used the Directed Acyclic Graph (DAG) as an intermediate form because the DAG represents control-flow information through its branches and joins, and provides effective traversal mechanisms.

We developed two tools by extending our framework: PARSEWeb and Hotspotter. We showed the effectiveness of our framework by comparing the results of developed tools with the results of related tools.

[1] `http://gonzui.sourceforge.net/links.html`
[2] `http://www.google.com/codesearch`

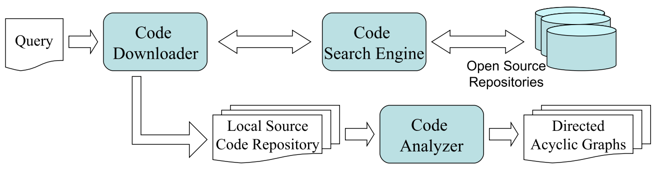

Figure 1. Overview of the framework

2. Framework

Our framework consists of three major components: the code downloader, code search engine, and code analyzer. Figure 1 shows an overview of all components and flows among different components.

Given a query, the code downloader interacts with a CSE and downloads relevant code samples. The downloaded code samples, which are partial, form a local source code repository that serves as input to the code analyzer. The code analyzer performs static analysis over each code sample with heuristics and transforms the sample into an intermediate form represented as a DAG. During transformation, statements inside loops like *while* and *for* are treated as a group of statements that are executed either once or not. While constructing DAG, the code analyzer also performs method inlining by replacing method invocations of the current class with the body of the corresponding method declarations. Each node in the constructed DAG represents a statement in the code sample. During transformation, the code analyzer uses several heuristics to gather additional type information for each statement. For example, the additional type information for a method-invocation statement includes the receiver object type, the return object type, and argument types. We explain a heuristic used in our framework through the code sample shown below:

```
public QueueSession test()   { ...
```

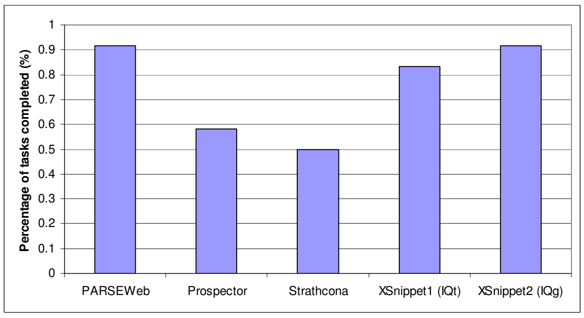

Figure 2. Evaluation results of PARSEWeb with related tools Prospector, Strathcona, and XSnippet

```
        return connect.createQueueSession(false,int);}
```

In this code sample, the method-invocation `createQueueSession` is a part of the return statement. The receiver object type of method-invocation `createQueueSession` can be inferred by looking up the declaration of the `connect` variable. But as our framework deals with the code sample that is partial, it is difficult to get the return type of the method-invocation with out being able to access the corresponding method declaration in the downloaded file. However, the return type can still be inferred from the return type of the enclosing method declaration. As the enclosing method declaration has return type `QueueSession`, we can infer that the return type of the method-invocation `createQueueSession` is `QueueSession` or one of its subtypes. In our framework implementation, we used GCSE as an underlying CSE.

3. Tools

We developed two tools by extending our described framework. The results of these tools show the effectiveness of our framework.

3.1 PARSEWeb

A common problem faced by programmers while reusing existing frameworks or libraries is that the programmers often know what type of object that they need, but do not know how to get that object with a specific method sequence. We developed PARSEWeb to address the preceding issue. PARSEWeb[3] accepts a query of the form "*Source* $\rightarrow$ *Destination*", and gives the *Source* and *Destination* object types as input to the described framework, which in turn provides graphs (DAG) for each downloaded code sample as output. PARSEWeb identifies nodes that contain the given *Source* and *Destination* object types as Source and Destination nodes, respectively. PARSEWeb extracts a method-invocation sequence by calculating the shortest path between Source and Destination nodes. We evaluated PARSEWeb with related existing tools Prospector [2], Strathcona [1], and XSnippet [3] for 12 specific programming tasks taken from the XSnippet approach. The results of our evaluation are shown in Figure 2. The XSnippet1 and XSnippet2 entries show results with two query-type techniques IQ_T and IQ_G of XSnippet, respectively. PARSEWeb performed better than Prospector, Strathcona, and XSnippet1. The results of PARSEWeb are at par with XSnippet2. Moreover, as PARSEWeb is developed based on our framework that uses CSEs for gathering relevant code samples on demand, unlike other related tools, PARSEWeb is not limited to the queries of any specific set of frameworks or libraries.

3.2 Hotspotter

Object-oriented frameworks are designed mainly with the concern of reusability. Framework users, who develop concrete applications by customizing frameworks, must be aware of its areas of flexibility that are commonly referred as hotspots. We developed Hotspotter for identifying hotspots in a given input framework. These hotspots can serve as a starting point for users of the input framework. Hotspotter also assists framework developers by identifying weakspots and deadspots. Weakspots are APIs that are rarely used and deadspots are APIs that are not used among the samples gathered from CSEs. These weakspots and deadspots can help developers to analyze whether to keep those APIs in further versions or whether the functionality provided by those APIs is subsumed by other APIs.

S.No.	Subject	Total	Hotspots		Weakspots		Deadspots	
	Name	No. of APIs	No.	%	No.	%	No.	%
1	JUnit	379	22	5.8	83	21.9	124	32.7
2	Log4j	1334	21	1.6	180	13.5	746	55.9
3	BCEL	2691	18	0.7	486	18.1	1867	69.4
4	Struts	4679	28	0.6	0	0	4332	92.6

Table 1. Evaluation results of Hotspotter with four subjects `Total: Total number of APIs, No.: Number of APIs identified as hotspots, weakspots, and deadspots and their percentage (shown with %) among the total number of APIs.`

Hotspotter identifies hotspots, weakspots, and deadspots in the given input framework by calculating the Usage Percentage (*UP*) of each public or protected API among the code samples returned by the CSE. The *UP* of an API is calculated as (`No. of usages of the API` / `Total no. of usages of all APIs`) * 100. Hotspotter uses two threshold values: Upper Threshold (*UTH*) and Lower Threshold (*LTH*). APIs with *UP* more than *UTH* are classified as hotspots and APIs with *UP* less than *LTH* and greater than zero are classified as weakspots. APIs with *UP* of zero are classified as deadspots. In our implementation, we used *UTH* value as 1% and *LTH* value as 0.1%. We plan to find optimum values for these parameters by analyzing other open source frameworks. Table 1 shows preliminary results of Hotspotter with four subject frameworks JUnit, Log4j, BCEL, and Struts. Our results show that the framework users often use only a small subset of the total number of APIs provided by the framework.

4. Conclusion

To exploit code search engines for several tasks that can help improve the productivity of programmers, we developed an extensible framework that helps to use the strength of code search engines and addresses the major limitations in using CSEs for those tasks. Our framework includes several heuristics to deal with partial code samples returned by CSEs. We showed the effectiveness of our framework with two tools developed based on our framework. In future work, we plan to extend our framework for detecting bugs by mining specification patterns from the code samples returned by CSEs. As the current implementation is dependent on the code samples gathered through GCSE, we plan to extend our framework to collect samples from other CSEs like Krugle and Koders to analyze results across different CSEs.

References

[1] R. Holmes and G. Murphy. Using structural context to recommend source code examples. In *Proc. of ICSE*, pages 117–125, 2005.

[2] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. of PLDI*, pages 48–61, 2005.

[3] N. Sahavechaphan and K. Claypool. XSnippet: Mining for sample code. In *Proc. of OOPSLA*, pages 413–430, 2006.

[3] Available at `http://ase.csc.ncsu.edu/parseweb/`