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

Denys Poshyvanyk[1], Mark Grechanik[2,3], and Collin McMillan[1]

[1]Computer Science Department, College of William & Mary, [2]Accenture Technology Labs, [3]Computer Science Department, University of Illinois, Chicago

Abstract

When programmers develop or maintain software, they instinctively sense that there are fragments of code that other developers implemented somewhere, and thus 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 (S3)* 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 the cryptographic concept).

Background

The three main problems inhibiting 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 a design and low level implementations.

State-of-the-art code search engines, such as Google Code Search, match words from search queries to the identifiers or comments in open-source projects. Unfortunately, these engines provide no guarantee that found code snippets implement concepts or features described in queries.

Programmers use third-party API calls to implement high-level requirements. Rather than attempting to directly map user queries to source code elements, we aim at connecting queries to usage documentation, then documentation to API calls, and then those calls to the relevant parts of source code based on which elements contain those calls.

We propose unifying *searching*, *selecting*, and *synthesizing* into a single framework (the S^3 architecture) based on the common abstraction and behavior-specific compositional mechanisms of software systems (e.g., API usage).

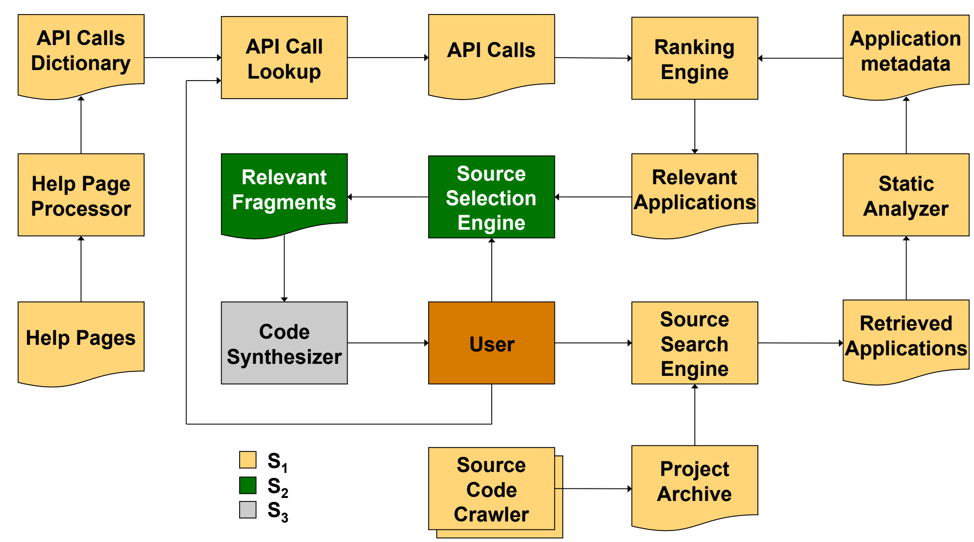


Figure 1. An overview of the S^3 architecture. The S_1 component searches for relevant applications from source code repositories, S_2 selects relevant fragments of code from those applications (at varying granularity), and S_3 synthesizes those relevant fragments into the user's code at his or her discretion.

S_2 Essentials

As a preliminary step, we tested an implementation of the S_2 component.

We focused on the official Java API documentation and 40 publicly available Java examples[1]. By using the given example descriptions as an oracle for mapping the user queries to source code fragments, we were able to compute *accuracy*, *discovery*, and their harmonic mean.

$$accuracy = 104 - n * (rank\ of\ correct\ result)$$

$$discovery = \frac{(queries\ with\ positive\ accuracy)}{(total\ number\ of\ queries)}$$

[1]http://www.java2s.com/

S^3 Walkthrough

In S_1, help pages are processed to associate text documentation to API calls. These are then linked to user queries with a natural language processing technique. A ranking engine combines this information with programs retrieved from a code search engine. The progress on indexing open-source software is presented in Table 1.

The code search engine chooses relevant applications from the index created by our source code crawler using the same user queries. In this way, structural and textual search methods are combined. Relevant applications are then statically analyzed to retrieve metadata. Metadata contain dataflow and dependencies among API calls.

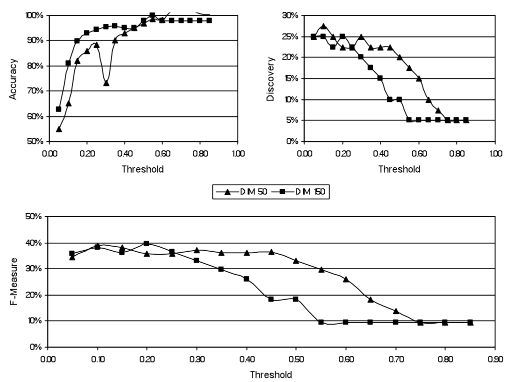
The ranking engine melds metadata with the lists of relevant API calls. Outputted is a list of the relevant applications which use the relevant calls.

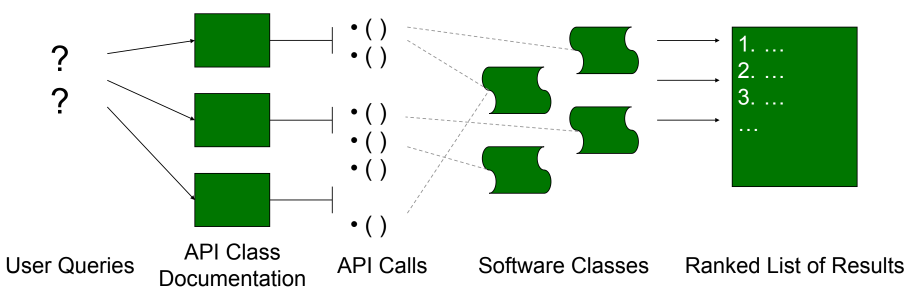
Given this set of relevant applications, the S_2 component selects portions implementing functionality described by user queries.

Figure 2 (below). An overview of the approach. We find the textual similarities between user queries and API documentation with Latent Semantic Indexing (LSI), filtering results with a similarity threshold.

Figure 3 (right). We ran every query through our system across the thresholds 0.05 to 0.85 in increments of 0.05. We then computed the average accuracy of all positive accuracies by looking up the correct result from the oracle.

Our system returns very high accuracy but relatively low discovery, typically providing the correct result within the top three answers, or not at all.





Source Code Crawler

Table 1. We are building and testing our own source code crawler for downloading, extracting and indexing open-source applications from repositories, such as Sourceforge.net.

Items	Count
Java Projects	21,934
Files	38,330
Files Downloaded (*.zip, etc.)	31,371
Files Skipped (*.exe, *.pdf, etc.)	6,959
GB Downloaded	105.62 GB
GB Skipped	45.71 GB
Files Indexed by Lucene	10,897
Java docs in index	100,866

Further Information

Visit Semeru: **http://www.cs.wm.edu/semeru/**
Email Denys Poshyvanyk: denys@cs.wm.edu
Email Mark Grechanik: drmark@uic.edu