Sandō: An Extensible Local Code Search Framework

David Shepherd
ABB, Inc.
Raleigh, NC, USA
david.shepherd@us.abb.com

Kostadin Damevski
Virginia State Universtiy
Petersburg, VA, USA
kdamevski@vsu.edu

Bartosz Ropski
Autodesk, Inc.
Krakow, Poland
bartosz.ropski@autodesk.com

Thomas Fritz
University of Zurich
Zurich, Switzerland
tfritz@ifi.uzh.ch

ABSTRACT

Developers heavily rely on Local Code Search (LCS)—the execution of a text-based search on a single code base—to find starting points in software maintenance tasks. While LCS approaches commonly used by developers are based on lexical matching and often result in failed searches or irrelevant results, developers have not yet migrated to the various research approaches that have made significant advancements in LCS. We hypothesize that two of the major reasons for this lack of migration are as follows. First, developers do not know which approach is the best, due to a lack of comparative field studies and the discrepancies in the underlying LCS process that these research approaches address. Second, developers lack access to a stable implementation of most of the research approaches. To address these issues, we studied a number of LCS approaches, distilled the general component structure underlying these approaches and, based on this structure, developed a LCS tool and framework, called Sandō. Currently used by developers at ABB, Inc. and elsewhere, Sandō also supports the flexible extension of its components to rapidly disseminate research advancements, and allows for user-based evaluation of competing approaches.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding, Tools and Techniques

General Terms: Design, Experimentation, Languages

Keywords: Code search, feature location

1. INTRODUCTION

Developers often rely on local code search tools to determine a starting point, or seed, in the code for software maintenance tasks [2]. In a recent study, 40 out of 48 instances of a developer performing a maintenance task began with the developer performing code search [3]. Due to the lexical matching approach that underlies the code search tools most commonly used by developers, these searches return an overwhelming amount of irrelevant results or no results at all, in most cases, and thus lead to failed searches [10].

While recent research on Local Code Search (LCS) has made significant advances, most developers still use failure-prone search tools based on lexical matching. We hypothesize that two of the major reasons for developers ignoring recent LCS research is, first, the lack of clear comparisons between approaches and, second, the unavailability of usable implementations. The difficulty in comparing research approaches on code search lies in the underlying differences of the approaches' internal process models. Even approaches that address the same part of the LCS process vary widely in aspects that are not essential to the research. For instance, two recent techniques, Promesir [4] and Shao *et al.*'s approach [9], address the reordering of the search results but are difficult to compare due to the differences in their splitting and word recommending algorithms, which are not the focus of the research work. In addition to the difficulty in comparing approaches, almost none of the research approaches for LCS provide a publicly available implementation [1] or are not necessarily usable by practitioners.

In this paper, we introduce the Sandō code search tool and framework that embodies a general and extensible LCS model. This LCS process model, (1) illustrates the differences between LCS approaches; (2) facilitates non-confounded comparisons of different approaches or the effect of a single component's improvement on the overall code search process; and, (3) allows for the integration of various approaches focusing on different aspects of the LCS process. Sandō is a standalone LCS tool implemented as a stable, open source Visual Studio extension. At the same time, Sandō serves as a framework that can easily be extended to quickly disseminate advances in research. Sandō complements existing lab-based LCS evaluation approaches (e.g., [1]) by enabling realistic, developer-based evaluations. Since Sandō's introduction on May 12, 2012, it has experienced a significant number of downloads [1] (> 400) and site visits (> 1000).

2. THE SANDŌ FRAMEWORK AND TOOL

At its core, Sandō uses a generic process model that we distilled by examining existing LCS approaches. To maximize the model's generality and provide best extensibility, we abstracted it from the three major categories of LCS approaches: information retrieval (e.g., [6, 5]), natural language (e.g., [10, 2]) and program analysis refined (e.g., [4, 9]) search based approaches. The resulting process model, shown in Figure 1, is composed of two major workflows, `Indexing` and `Querying`, consisting of eight major components.

Process Model for Local Code Search.

The `Indexing` workflow begins with the `File Monitor` component, which forwards modified or newly added ap-

[1] see `http://visualstudiogallery.msdn.microsoft.com/06f39a31-20ce-408c-afee-8a02b484db1c` for current statistics

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

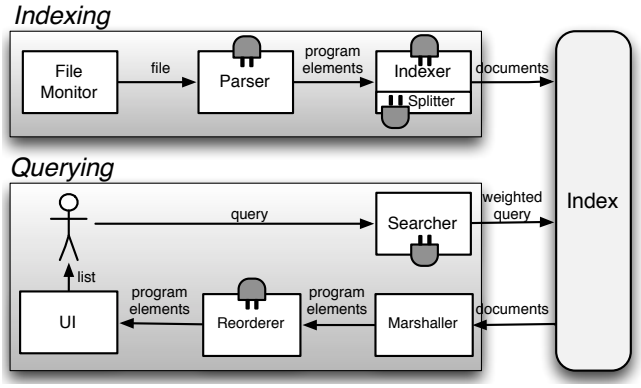

Figure 1: The general LCS workflow underlying Sando, consisting of two distinct phases: indexing and querying.

plication files, the raw data, to the `Parser` component. In turn, the `Parser` decomposes each file into a set of program elements, such as methods, fields, and classes, that are then translated into documents by the `Indexer`. Documents are entirely text-based representations of program elements, appropriate for text-based indexing. Internally, the `Indexer` uses the `Splitter` sub-component to ensure that the text, in particular the identifiers, is properly split into individual words (e.g., "`openFile`" → "`open`","`file`").

The `Querying` workflow contains the interactive part of the LCS process. The user's query is sent to the `Searcher` component, where it is rewritten according to a set of rules. This includes the addition of synonyms and/or weights to the query in order to, for example, prefer matches in method names over matches in method bodies. The rewritten query is then forwarded to the `Index` database, which returns a ranked list of documents that match it. The `Marshaller` processes these documents, reverting them into program elements, and the `Reorderer` can then be used to reorder the final search results according to any scheme specified (e.g., a call-graph-based page rank score). Finally, the ranked program elements are displayed to the user in the `UI` component.

Extension Points.

While the described process model is shared by many of the LCS approaches, the behavior of individual components can vary significantly between approaches. For instance, approaches vary in the splitting of identifiers, by taking into account camel-case [2] or not [6], or in the program elements that are accepted by the parser, e.g., C++ [2] or C, C++ and Java [6]. To support the most common variations, the Sando framework provides a set of extension points, one for each of the following five components: the `Parser`, the `Splitter`, the `Indexer`, the `Searcher` and the `Reorderer`.

The Sando Search Tool.

Besides being an extensible framework for building new code search tools, Sando is also a standalone LCS tool, implemented using the above process model. Sando provides default implementations for each extension point, namely, a `Parser` that handles C#, C++, and C at the granularity of method, field, class, comment, and property elements; an `Indexer` that separates elements into name and body fields; a `Splitter` that splits identifiers on camel-case, underscore, and numbers; an `Index` based on a vector space model using TF/IDF scoring; a `Searcher` that weighs element names more highly than element bodies; a `Reorderer` that does no reordering for now; and a `UI` that displays the results list.

Sando's extension point mechanism facilitates the quick creation of working LCS tools or the integration of new research ideas into a stable, open source tool that then also supports comparative field studies. The default implementation for each of the extension points can easily be replaced by providing another implementation of the provided extension point interface and updating a single configuration file. Thus, a researcher could use the Sando framework with its extension point mechanism to evaluate different types of word-splitters or a new LCS approach that reorders search results using, for example, execution trace information. Sando's documentation[2] details the extension process and describes default extension points.

Related Work.

As a standalone tool, Sando is most similar to information retrieval-based approaches (e.g., [6, 5, 7]), but differs by providing an extension point mechanism to support the integration of several approaches and the quick experimentation of new ideas in a stable and working tool.

Sando shares motivation with TraceLab [1], a very recently published framework for evaluation and comparison of feature location techniques. TraceLab provides an experimentation framework, including test data sets, to evaluate the precision and recall of different feature location techniques. Sando's framework is complementary in that it enables research approaches to be validated via realistic user studies or A/B testing. In addition, Sando contributes a vehicle to disseminate successful research approaches to developers via a working tool.

3. REFERENCES

[1] B. Dit, E. Moritz, and D. Poshyvanyk. A tracelab-based solution for creating, conducting, and sharing feature location experiments. In *Int. Conf. on Prog. Comp*, 2011.

[2] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Int. Conf. on Soft. Eng.*, 2009.

[3] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. on Soft. Eng.*, 2006.

[4] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Int. Conf. on Automated Soft. Eng.*, 2007.

[5] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *Working Conf. on Reverse Eng.*, 2008.

[6] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Working Conf. on Rev. Eng.*, 2004.

[7] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Int. Conf. on Soft. Eng.*, 2011.

[8] D. Poshyvanyk, M. Gethers, and A. Marcus. Concept location using formal concept analysis and information retrieval. In *Trans. on Soft. Eng. and Meth.*, 2010.

[9] P. Shao and R. K. Smith. Feature location by IR modules and call graph. In *SE Regional Conf.*, 2009.

[10] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. V. Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Int. Conf. on Aspect-Oriented Soft. Dev.*, 2007.

[2] `sando.codeplex.com/wikipage?title=Custom%20Sando%20Extensions`

APPENDIX

A. DEMONSTRATION SCRIPT

Here we demonstrate Sando's utility as both a standalone code search tool as well as a research-enabling framework. First, we use a realistic maintenance scenario to show the advantage that Sando provides over a state-of-the-practice tool to find relevant code. We then pose a sample research question and show how Sando's extension point framework can be leveraged to quickly implement a tool that would allow researchers to investigate this question. Finally, we briefly describe an implementation of an approach from a recent research paper in Sando.

A.1 The Sando Search Tool

A.1.1 Maintenance Scenario

The Notepad++ [3] program is an open source, C++ project that consists of approximately 350 source files and 18KLOC. It is described as a "source code editor and Notepad replacement that supports several languages." As such, it contains many familiar document editing functions, such as printing files, highlighting code syntax and displaying line numbers.

Notepad++ has been under development for several years and thus has a backlog of open bugs. In one of them, a user has reported that Notepad++ will cut off its line numbers, which are displayed in the left margin of an open file, under certain circumstances[4]. In this demonstration we will focus on finding the code relevant to fixing this bug.

A.1.2 Using Lexical Search

When beginning to fix a bug developers often use code search tools to locate a starting point in the project for this task. Unfortunately, when using state-of-the-practice tools it is very difficult to craft an effective search query and retrieve a good starting point. Consider a developer using FIND IN FILES, a regular expression-based code search tool commonly used in Visual Studio in the beginning of fixing the Notepad++ bug. If this developer searches for *linenumber*, 322 results are returned. Because these results are unranked, the developer is very unlikely to review them manually, as relevant results may not appear until near the end of the list. Developers faced with this scenario will most likely refine their initial query to reduce the number of results. A common way of reducing the number of regular expression search results is to add terms to the query. If the developer expands his query to *linenumber*update* no result set is returned because regular expression searches are sensitive to word ordering. However, if the developer switches his term ordering and searches for *update*linenumber* then the correct result, method `updateLineNumberWidth`, is returned as second result. Unfortunately, the developer had to put considerable mental effort and time into crafting an effective search query.

A.1.3 Using Sando Search

If the developer uses Sando to find a starting point to fix this bug he or she can create an effective search query with much less effort (see Figure 2). For instance, each of the queries: *linenumber update*, *update linenumber*, *line number update*, *line number width*, or even *line width* would result in the retrieval of relevant code. These queries would find the relevant method ranked as result number 1, 1, 2, 1, and 2 respectively. In contrast, using the same queries (with spaces replaced by asterisks) in FIND IN FILES results in an empty set, a set of three, an empty set, a set of eight, and a set of 72 unranked results respectively. Thus, for this set of queries Sando always returns the correct result in the top two ranked slots whereas FIND IN FILES essentially fails to provide a small set of relevant code for three of the five queries [5].

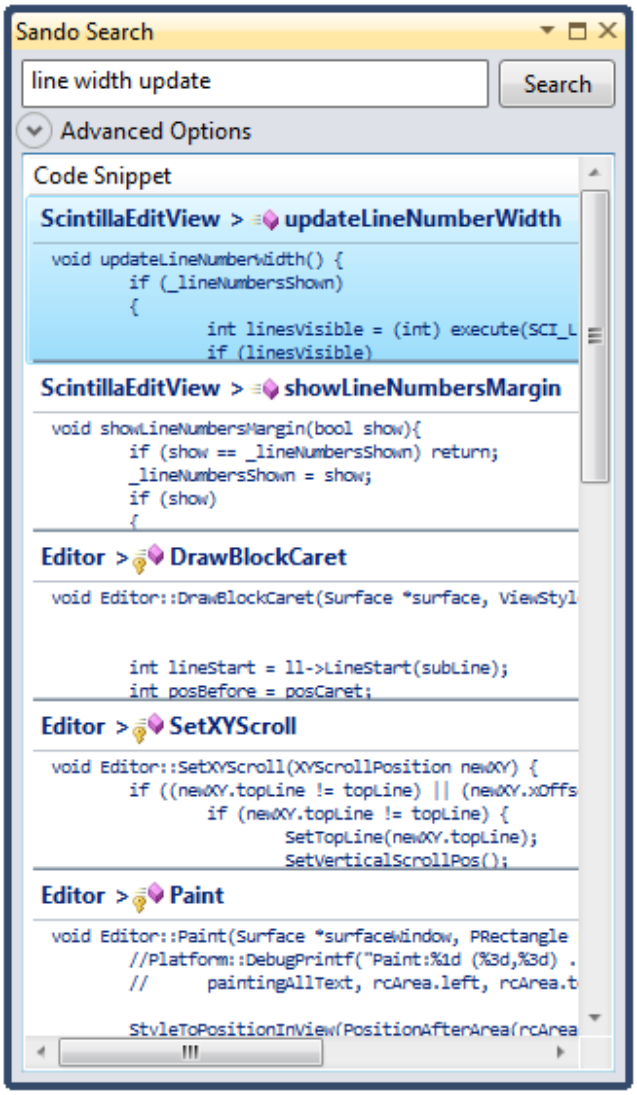

Figure 2: Using Sando, a higher percentage of queries are successful. The presented search for *line width update* returns the most relevant method as first result.

[3] `http://notepad-plus-plus.org/`

[4] `http://sourceforge.net/tracker/?func=detail&aid=3513126&group_id=95717&atid=612382`

[5] A large set of unordered search results are often considered a failure by users, who will try to refine their query instead of inspecting results [8]

A.2 Sando Extensions

A.2.1 Extension Scenario

While Sando users can create search queries with much less effort, developers still have to be careful to spell query terms correctly. If a developer searches using a misspelled word, such as *find curent* Sando will fail to return any results in Notepad++, as the misspelled term *curent* does not exist in this code base. Therefore, researchers may want to investigate the following sample research question:

> **ExRQ:** Would the auto-correction of query term spelling increase the effectiveness of code search?

While this ExRQ may seem to have an obvious answer (i.e., auto-correction should improve performance), the particularities of source code may confound common sense. For instance, in many projects abbreviations or intentional misspellings are used, such as *calc* for *calculate*. In this case, automatically correcting *calc* would likely fail to retrieve the relevant results.

A.2.2 Creating an Extension

To investigate this sample research question on actual code with real users as a lab study or as a field study, researchers can use Sando's extension point mechanism to easily implement an auto-correcting version of Sando. This version of Sando would automatically correct misspelled search terms prior to executing the search. In order to do so, researchers can use the Searcher extension point, specifically, by implementing the `IQueryRewriter` class. Below, we provide a listing of `RewriteQuery`, the only method that must be implemented (apart from initializing the data structures) to enable the auto-correction extension. This method takes a query as input (line 1), splits it into words (line 4), checks if the word is misspelled (line 6), and then replaces the word with the first suggestion in case it is misspelled (line 9).

```
1  public string RewriteQuery(string query)
2  {
3    Initialize();
4    var queryWords = query.Split(' ');
5    foreach (var queryWord in queryWords) {
6      if(!engine["en"].Spell(queryWord)) {
7        var suggestions = engine["en"].Suggest(
             queryWord);
8        if(suggestions.Count>0) {
9          query = query.Replace(queryWord, suggestions.
               First());
10       }
11     }
12   }
13   return query;
14 }
```

In addition to implementing the `IQueryRewriter` interface, a researcher must also update the extension point configuration file. The necessary steps consist of adding an entry in the configuration file (line 1), specifying the new query rewriter class (line 2) and the relevant library (line 3), and then copying the .dll file into the extension directory.

```
1  <QueryRewriterConfiguration>
2    <FullClassName>Sando.ExperimentalExtensions.
         SpellChecking.SpellCheckingQueryRewriter</
         FullClassName>
3    <LibraryFileRelativePath>ExperimentalExtensions.
         dll</LibraryFileRelativePath>
4  </QueryRewriterConfiguration>
```

Once Visual Studio is restarted Sando will appear to work as per usual. However, performing the same search as before, for "find curent", results in the activation of the auto-correction extension, which corrects the query and returns the relevant results. Assuming a reasonable implementation of the extension, Sando will continue to be a robust implementation of code search, suitable for lab and even field studies.

A.3 Implementation of Existing Research

In the previous example we replaced a Sando component to investigate a somewhat trivial approach, which was not previously studied. However, because Sando offers a number of extension points, many different LCS approaches can potentially be implemented (and thus investigated). To demonstrate Sando's capacity to implement existing research scenarios, we implemented an approach similar to Shao *et al.* [9] in Sando. This approach combines information retrieval scores with call graph information to reorder the search results. In the case study conducted by the authors, this approach works well, although several threats to the generality of the results are noted.

To implement the approach presented by Shao *et al.*, we used the Reorderer extension point. For the implementation, we only had to write a total of 73 LOC and immediately had a working LCS tool which we used to search Notepad++. In our preliminary investigation, this approach seemed promising, effectively boosting search results that were highly connected to other search results. However, after further investigation we also noticed that getters and setters, which are called by many methods, were often boosted too high in the search results and that special rules for getters and setters might improve the performance of the research approach. The short time between research idea and fully working code search tool allowed us to quickly gather feedback in a realistic setting, and can ultimately result in an improved approach.

As seen in this example, Sando allows us to quickly implement and investigate existing research approaches. To further extend the number of such approaches that can be implemented, we intend to add additional extension points to Sando in the future. One such extension point that we plan to add in the near future will allow for the replacement of the Index component, enabling the use of different indexing algorithms.

A.4 Tool Availability

Sando's latest release is available on Visual Studio Gallery (`http://visualstudiogallery.msdn.microsoft.com/`) as a VSIX file. Opening this file on a Windows machine will automatically install Sando into Visual Studio 2010. Once Visual Studio has been launched the Sando Search View can be found under *View* > *Other Windows* > *Sando Search*. Sando's source code is also available as open source on CodePlex (`http://sando.codeplex.com`).