Building Bing Developer Assistant
MSR-TR-2015-36

Yi Wei · Nirupama Chandrasekaran · Sumit Gulwani · Youssef Hamadi
Microsoft Research
{yiwe, niruc, sumitg, youssefh}@microsoft.com

May 12, 2015

Abstract

Software developers heavily rely on code snippets and API usage examples searched on the Internet. This paper presents Bing Code Search, a Visual Studio extension that allows developers to write, within an IDE, free-form natural language questions, and get C# code snippets answering those questions. Bing Code Search automatically adapts the suggested snippets into the user's programming context via variable renaming, and records users' interactions to improve its suggestions. Compared to prior related research, Bing Code Search provides a complete automation of the full search-paste-adapt process. Three weeks after we released this free extension, more than 20,000 users downloaded it; and they issue on average 3,000 queries per day.
We believe that Bing Code Search is the most widely used tool in its category. In the following, we fully describe our framework, and draw clear empirical evidence of the benefits of Bing Code Search: (1) From our evaluation benchmark, compared with Bing's result, Bing Code Search delivers more relevant snippet solutions. (2) In a controlled experiment, it was able to save developers 28% of time on completing API related tasks. (3) Telemetries collected from thousands of users show some users already built up the habit of using the tool: they issue multiple queries to solve a complex task, or use it as a fast auto-completion.

0.1 Introduction

Today, developers rely on programming resources found on the Internet, such as tutorials, API usage examples, or other code samples, to complete their daily tasks. The *search-paste-adapt* approach that they follow has the following drawbacks:

- *The developer loses productivity*: she has to switch from IDE to browser and back, keep focus while searching the web for relevant pages, scan these pages to find snippets, copy selected text fragments into the IDE, and adapt them to her specific problem and context.
- *The search misses some signal*: the browser does not have access to the contextual information available in the IDE, resulting in less precise search results.

We developed a framework called Bing Code Search that aims to address these issues. Bing Code Search is a plugin in Visual Studio; it works as follows: developers describe the task they want to complete in natural language within the IDE. Bing Code Search performs the search-paste-adapt process automatically and suggests relevant and adapted snippets: specifically, it searches through the web for candidate snippets, re-ranks the result using code-specific features, then adapts the snippets to the user's programming context by renaming variables before presenting them to the user. It further monitors snippet selection by users, and improves suggestion quality based on this feedback.

There is significant recent research on integrated code search. Our principal contribution is a widely used[1], fully automated pipeline to improve search-paste-adapt operations in a programming environment. Our secondary contributions are the following:

- A set of language-independent features to re-rank code snippets.
- A language-independent algorithm for code adaptation through variable renaming.

We thoroughly evaluated Bing Code Search. Before shipping the extension to a large public, we performed internal experiments and user studies. Three highlights are:

- Our re-ranking and integrated pipeline results in a 4.6% increase over Bing's search result in precision, measured by NDCG at rank 1 and 15.8% increase at rank 10. This improvement over a commercial search engine is significant, and results in a noticeably more relevant snippet list. We are working with Bing to transfer the code ranking technique into their infrastructure.
- The variable renaming algorithm is able to propose correct renaming in 71% of the situations.
- In a user study we found Bing Code Search was able to save users 28% time on completing API-related tasks.

[1] **http://visualstudiogallery.msdn.microsoft.com/a1166718-a2d9-4a48-a5fd-504ff4ad1b65**

Three weeks after the release, more than 20,000 users have downloaded this free extension and they issue on average 3,000 queries per day. We believe that Bing Code Search is the most widely used tool in its category. Telemetry data coming from this large users base allowed us to explore the practical benefits of integrating code search in an IDE. We came with the following general findings:

- Users immediately understand the benefit and usage of the extension. This is important to grow a large user commuity.
- Users are excited about this feature in their IDEs. Many users left positive feedback; many request us to support more languages and different IDEs.
- Some users already formed using habits: we observe some users issue a set of related queries in a short period of time, trying to solve a complex task in multiple steps. We also observe some other users issue the same query on different days, showing they are using the tool as a faster, task-level auto-completion.

These findings indicate that Bing Code Search can benefits different user groups: beginner and intermediate level developers can use it to program against APIs. Expert developers can use it as a high level auto-completion tool. Users recently moved from one language to another can use it to get a quick start.

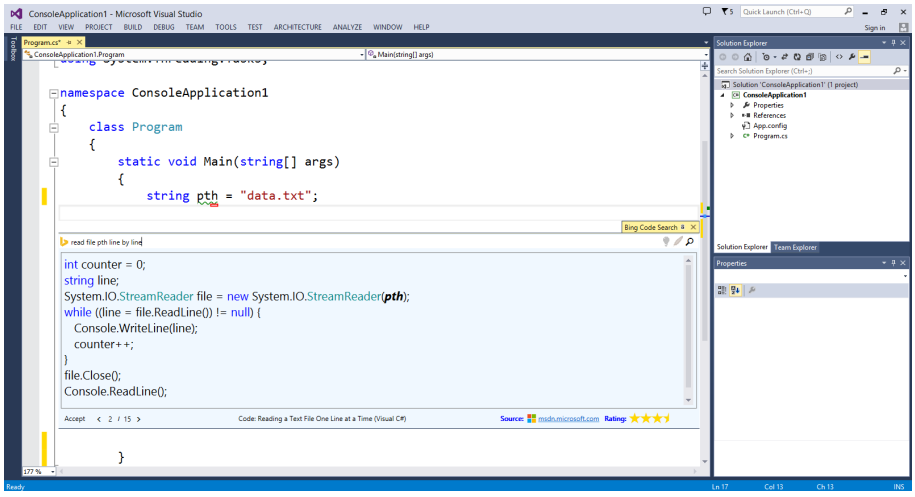

Figure 1: The code suggestion interface

0.2 Usage Scenario

This section walks through how a user interacts with Bing Code Search. We consider a user working with the C# language in the Visual Studio environment who wants to find out how to read a text file line by line.

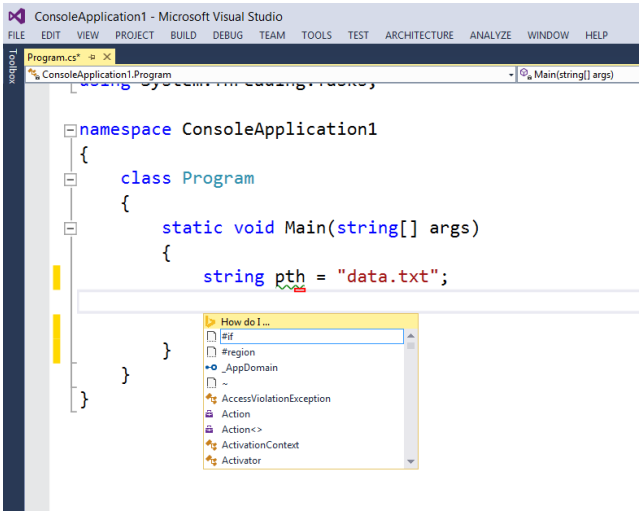

Figure 2: Triggering Bing Code Search.

The user triggers IntelliSense™ and selects the *How do I ...* list item. See Figure 2. This activates the Bing Code Search window, where the user can directly type its *"read file pth line by line"* query. This is presented in Figure 1. The query can reference variables that exist in the scope. Here, the variable `pth` holds the name of the file to read. We found that it is natural to reference variables when formulating a query, and an advantage of these references is that they clearly identify which parts of the current context are related to the task.

After pressing `Enter`, the user is exposed with a ranked list of snippets that can be navigated through. The snippets are retrieved from the web and re-ranked according to their relevancy to the query and their code quality (Section 0.4).

Each snippet comes with meta information about its origin (Source), and when available quality (Rating). This is on the bottom right of the window. These information are helpful to evaluate the credibility of the snippet.

To make the snippet suggestion easier to understand, Bing Code Search replaces variables in the snippets with the ones mentioned in the user query through variable renaming (Section 0.5). For instance, the variable `pth` in the above query is used in the third line of the suggested snippet. After the user identifies a desirable snippet, she can press `Enter` to insert the code into the editor.

Bing Code Search records which snippet is selected.This feedback is used to improve the results for future similar queries and integrate in its ranking the user selections (Section 0.6). For example, if in the future some user asks a query similar to *"read file one line at a time"*, she will be exposed with snippets where previously selected ones will be pushed to the top of the list.

We provided an online version of Bing Code Search at **`http://codesnippet.research.microsoft.com`**. A video from the same page shows the extension in use.

0.3 Related work

There is substantial recent research on answering queries from programmers. We discuss four areas related to work in this paper.

0.3.1 IDE-based auto-completion systems

The Blueprint [1] tool by Brandt et al. suggests snippets from users' natural language queries. The authors showed that by integrating web search into the Adobe Flex Builder IDE, users could perform tasks faster. Blueprint harvests snippets from the web and uses Adobe Community Help search APIs and a Google Custom Search engine to offer suggestions. One essential metric of such a recommender system is the precision of the suggestions (Recall is not as important in such a setting because a suggestion list shown to the developer does not aim to include all relevant snippets). The Blueprint paper does not report any metric of the suggestion list. Since the snippets are from web pages, in case there are several snippets from the same page, deciding their display order is non-trivial. If the tool displays all the snippets, the suggestion list would include many irrelevant snippets since it is common that only a small amount of snippets from a web page are relevant to the user query. Another question that is not answered in this prior work is a comparison against commercial search engines. These are all practical questions that we solved during the design of Bing Code Search. Blueprint does not perform context adaptation of the suggested snippets; users need to select relevant lines from the suggested snippets and copy-paste them into their editors. Bing Code Search, on the other hand, offers context adaptation through variable renaming and user edit collection, all of which are fully automated.

The SnipMatch tool [28] by Wightman et al. goes beyond the Blueprint tool to suggest snippets with variable renaming from user queries. It does so by requiring users to provide the snippets. Those curated snippets contain descriptions used for matching against user queries and placeholder variable specifications used for variable renaming. SnipMatch requires much higher user involvement than Bing Code Search. We think this high involvement limits its usability. The variable renaming algorithm in SnipMatch only relies on variable types. The algorithm does not support the case where there are several variables of the same type in either the query or the snippet. In such cases, further distinctions are needed to decide which variables to rename. The variable renaming algorithm in Bing Code Searchstarts with type-correct bindings, and uses other information, such as variable concept and variable def-use relations to perform the renaming.

Little et al. [11] presented the keyword programming method, which can *expand* a list of user-given tokens into a valid expression that may use variables in context. Keyword programming focuses on suggesting a single expression, while Bing Code Searchcan suggest larger snippets as well.

The Codelets tool [17] by Oney et al. can be used to produce interactive documentation. Interactive documentation supplements a piece of code with a dialog interface for users to adapt the code, for example by specifying actual parameters, and choosing methods to call. These documentations are only available if someone spends the effort to produce them; in contrast Bing Code Searchperforms adaptations in a completely

automated way, but it is restricted to simpler adaptations.

The Sando tool [22] by Shepherd et al. provides a framework to find code using a vector space text search engine. It helps programmers create search terms to match identifiers such as class and method names more easily. Sando focuses on locating existing code, while Bing Code Searchfocuses on suggesting new code, so there is no component in Sando to rank code according to code quality or to perform code adaptation.

0.3.2 Code repository mining

Code repositories represents valuable resource to learn how people program. The Jungloid tool [13], the XSnippet tool [21], the PARSEWeb tool [25] and the MatchMaker tool [30] focus on mining code for creating objects of a given target type.

The GraPacc [15] tool by Nguyen et al. and the UP-Miner [26] tool by Wang et al. mine API usage patterns. Similar to them, the TIKANGA tool [27] by Wasylkowski et al., the PRIME tool [14] by Mishne et al. and the GK-tail tool [12] by Lorenzoli et al. learn automata from existing code repositories. The mined patterns or automata can be used to suggest follow-up statements based on the user's incomplete code, without requiring the user to type a query. This provides the so called *zero-query* experience. Pattern-based code auto-completion tools work when the user knows which API to use, while Bing Code Searchcan handle the situation when the user does not know the name of the API to use.

0.3.3 Code search tools

Multiple tools exist for code search, such as Ohloh (originally named Koders) [16], and the closed-down Google Code Search (GCS) [5]. The usability of such tools is quite limited because a user needs to know the right keywords.

Reiss et al. [19] developed a tool that allows a user to apply a set of keywords along with a test suite to search for code: the keywords are used to match snippets syntactically; the test suite validates the snippets semantically. However, providing a test suite is a lot of work in the common case where a user simply wants to know about an API.

The SNIFF tool [3] by Chatterjee et al. suggests code by first translating method calls in a snippet into their corresponding API documents, and using text-matching algorithms to relate snippets to user queries.

Most people use a general-purpose search engine when facing programming difficulties. Providing better results than mature commercial search engines, even in specialized categories such as code, is very difficult. None of the related work reports a comparison against commercial search engines. Bing Code Searchbuilds its suggestions on top of the Bing search engine and brings significant improvements to both the Mean Reciprocal Error Rank (MRR) metric and the Normalized Discounted Cumulative Gain (NDCG) metric, both of which are widely used to evaluate information retrieval systems. Besides, Bing Code Searchprovides an altogether more integrated search experience.

Table 1: Features in snippet ranker

Feature name	Type	Description
Url position	real	the rank (the first page has rank 1, and so on) of the web page where the snippet is extracted
First/Second/Third url	binary	1 if the snippet is from the web page of rank 1/2/3; 0 otherwise
In-page order	real	index of the snippet from a web page. The first snippet in a page has index 1, and so on
First in-page	binary	1 if the snippet appears as the first snippet from the web page
Clickthrough	real	larger value indicates the snippet mentions more APIs which correlate to the user query
Tf-idf	real	larger value indicates the snippet mentions more APIs common to all snippets for the user query
Snippet length	real	the number of lines of the snippet
API calls	real	the number of API calls in the snippet
Unknown API calls	real	the number of API calls that are unknown
Unknown types	real	the number of types referenced in the snippets that are unknown
Compilation errors	real	the number of compilation errors for the snippet

Commercial search engines perform query expansion to boost retrieval performance. To the best of our knowledge, those query expansion algorithms are not optimized for code search. By expanding user queries with more software related words, which can be calculated by methods such as the one proposed by Yang et al. [29], Bing Code Searchmay get better results.

0.3.4 Program synthesis

Program synthesis accepts specifications and generates code fulfilling those specifications. It is related to our work: our framework takes user queries as specifications and suggests snippets. Solar-Lezama et al. [23] used *sketches* to synthesize programs. A sketch is a program with placeholders. Their synthesizer takes a sketch along with a reference implementation and generates an optimized version of the reference implementation. Srivastava et al. [24] introduced a technique that interprets program synthesis as generalized program verification. Gulwani [6] proposed a technique that synthesizes Spreadsheets programs from a set of input-output examples. Compared to our framework, these synthesis tools, which are based on logical reasoning, deliver results of higher quality: they guarantee a solution if there exists one. But the specification effort required to use such tools is much higher than just typing a natural language query in the editor. Recently, Le et al. [9] have described a natural language based interface to synthesizing smartphone automation scripts based on natural language understanding and program synthesis. However, all of this prior work in the area of program synthesis is mostly domain-specific and hence limited in its applicability.

0.4 Re-Ranking snippets

Starting from this section, we describe how Bing Code Searchworks.

Given a user query, Bing Code Searchfirst prepends it with the word "c#" and then uses the Bing search engine to find web pages related to that query. Upon retrieving the top 20 web pages, Bing Code Searchextracts text from four kinds of HTML tags: $\langle pre \rangle$, $\langle code \rangle$, $\langle p \rangle$, and $\langle div \rangle$. Most text fragments do not contain code. The framework uses the Microsoft Roslyn C# parser [20] to identify code out of text. The Roslyn parser can overcome certain syntax errors. This proves convenient when handling code from

the web, because many code fragments do not strictly follow the C# grammar syntax. The parser generates syntax trees for code fragments. The framework uses these syntax trees to identify various entities such as classes, methods, and variables.

Bing Code Searchneeds to rank the extracted snippets before showing them to the user. The quality of ranking is essential to the user experience and is a core metric for such recommendation systems. A first natural idea would be to put on top of our list all snippets from the most highly ranked web page, in their order of appearance within the page, then all snippets from the second page, and so forth. We found that this ordering is not satisfactory. The reason is that search engines focus on document-level relevancy whereas Bing Code Searchneeds to evaluate the relevancy of individual snippets. It is common that a web page contains multiple snippets, of which few are relevant. Therefore, taking all snippets from a page without discrimination typically results in a low precision.

To improve the quality of snippet ranking, we set out to build a machine learning based ranker for snippets. The desirable properties of the ranker are:

- It should take into account the *search engine ranking*: search engines already integrate sophisticated ranking techniques. Their reported ranking should be improved upon, but not be discarded.
- It should favour snippets that have a high *code relevancy* to the user query.
- It should favour snippets that have a high *code quality*.

Following a traditional machine learning process, we first formalize the relation between user query and snippets as a set of features and then train a ranker from labeled data. The features we are using are listed in Table 1; they are designed to address the three aforementioned properties. Every single feature captures part of the intuition that we give to the ranker. For a particular snippet, some features may fail to identify good code, or may even not apply, but collectively, they try to push relevant snippets to the top.

0.4.1 Original rank preserving features

In the table, the first group of features (*Url position* to *First in-page*) captures the original ranking from the search engine. The *Url position* feature captures the rank from the search engine. The *First/Second/Third* url features are derived features from Url position; they highlight snippets from top pages returned by a search engine. The *In-page order* feature captures the position of the snippet within a web page. The *First in-page* feature derives from *In-page order* feature and gives more weight to snippets appearing at the beginning of a web page since, empirically, relevant snippets tend to appear at the top of their containing web pages.

0.4.2 Code relevancy features

The second group of features (*Clickthrough* and *Tf-idf*) tries to model code relevancy. The *Clickthrough* feature models the mapping from user queries to C# tokens in snip-

pets. The *Tf-idf* feature models C# token popularity among all snippets retrieved for a particular query.

In addition to these features computed on the documents, search engines traditionally use another indicator of relevance, computed from online usage: they record the url u that is clicked on for every query Q, gathering pairs $\langle Q, u \rangle$ that are called clickthrough data. The same idea is used in Bing Code Search, in our case u represents the selection by a user of a specific snippet.

Bing Code Searchtreats every snippet as a sequence of C# tokens and calculates, for a token t, the probability that t occurs given the user query, that is, $P(t|Q)$. Let $Q = q_1, q_2, \ldots, q_n$ be the set of a words in a given query. The probability of a C# token t occurring given the query is defined as

$$P(t|Q) = P(t|q_1, q_2, \ldots, q_n) = \sum_{i=1}^{n} P(t|q_i) \cdot P(q_i|Q) \tag{1}$$

where $P(q|Q)$ is the unsmoothed unigram probability of the term q in the query Q. It quantifies how likely the query term q appears in queries:

$$P(q|Q) = \frac{\alpha_q}{\sum\limits_{q' \in Q} \alpha_{q'}} \tag{2}$$

where α_q is the appearance frequency of q in all possible queries.

To estimate $P(t|q)$ for each C# token t in any snippet and each query term q, we used a standard procedure of training statistical word alignment models [2]. The result model maximizes the probability of generating C# tokens from user queries over training data. Our training data consists in 15 days of clickthrough records from Bing's query log. To estimate α_q, we continue using the same query log:

$$\alpha_q = \frac{\#times\ q\ occurs\ in\ query\ log}{total\ term\ count\ in\ query\ log} \tag{3}$$

With $P(t|Q)$ defined, the overall probability of a snippet S occurring given a user query Q is

$$P(S|Q) = P(t_1, t_2, \ldots, t_m|Q) = \prod_{j=1}^{m} P(t_j|Q) \tag{4}$$

where the snippet S consists of tokens t_1 to t_m and the tokens are treated as independent for simplicity. We use log-sum to avoid underflow in calculating the product of probabilities. This log-sum value is used as the value of the *Clickthrough* feature.

The *Tf-idf* feature captures the fact that, for a query, there usually exists a set of APIs that are used in the solution. This set of APIs are commonly mentioned in various snippets extracted for that query. We define the value of this feature based on the term frequency-inverse document frequency (tf-idf) statistic [8] for each C# token (such as keyword, class name, method name) in the snippets.

Tf-idf gives large values for tokens that appear more often in the extracted snippets for a given user query, such as `StreamReader` in the file reading example. At the same time, it gives small values to tokens that appear often across *all* snippets, such as

`new` and `string`. In order to calculate idf values, we collected 2.5 million snippets from 1.3 million web pages from popular C# websites, and use them to represent the set of all snippets. The value of the *Tf-idf* feature is calculated as the average tf-idf of the tokens appearing in a snippet. A larger relevancy value indicates that the snippet is more likely to be the solution.

0.4.3 Code quality features

The third group of features (*Snippet length* to *Compilation errors*) models code quality in both syntactical and semantical aspects.

Snippet length is the number of lines of a snippet. According to our experience, long snippets are less likely to be appreciated by programmers. *API calls* is the number of API calls in a snippet. Bing Code Searchfocuses on API related queries, so if a snippet does not mention any API, it is less likely to be relevant. These two features are syntax-oriented: their values can be calculated from snippet AST trees.

The features *Unknown API calls*, *Unknown types* and *Compilation errors* are semantic-oriented: to calculate their values, we need to compile snippets. The most challenging part in compiling snippets retrieved from the web is to decide proper namespaces for mentioned types/methods and the needed libraries. The same name can, additionally, come from multiple libraries, in which case we need to decide which one is more probable.

To recover the link from API names to namespaces, we analyzed source code from 12,000 open-source projects collected from GitHub, CodePlex and BitBucket. This code corpus contains 357 million lines of code, 3.1 million classes and 17 million methods. From each class file, we collected imported namespaces and mentioned type names. We then estimate the probability of the needed namespace N for a type T by calculating

$$P(N|T) = \frac{P(N,T)}{P(T)} = \frac{\#files\ where\ both\ N\ and\ T\ occur}{\#files\ where\ T\ occurs}$$

For each type T, we rank the namespaces by the conditional probability $P(N|T)$ and choose the first one as the namespace to import to compile the snippet.

Once the namespaces are decided, identifying the library is easy. The library names almost always appear as the prefix of the namespace. We use the library names to download the required Dynamically Linked Libraries (DLLs) from Nuget (the online collection of C# libraries). We always download the most updated version for a library.

With namespaces and library dlls sorted out, we use the C# compiler from the Roslyn toolkit [20] to compile snippets. Note that even with the imported libraries, a snippet may still fail to compile. The Roslyn compiler does its best to compile a snippet and, when it fails, reports error messages. From the error messages, we calculate values for the feature *Unknown API calls*, *Unknown types* and *Compilation errors*.

0.4.4 Training the snippet ranker

We use supervised learning to build the snippet ranker. To obtain training data, we sampled 150 queries containing the word "C#" from one month of Bing's query log. Those

queries were then sampled according to their appearance frequency in the query log, so common queries are more likely to appear in the sample than uncommon queries. This gives a realistic query population.

For each query, we asked a programmer to create a context by introducing necessary variables. An example context for the query *"split string"* is shown in Listing 4. We then used Bing to retrieve the top 20 web pages and then extract snippets from the returned web pages. For each extracted snippet, we manually annotated whether it is relevant to the query. The goal was then to train a ranker which ranks snippets that are annotated as relevant on top of the ones annotated as irrelevant using the designed features. Many supervised ranker learners [10] are available, and we used the FaskRank algorithm [4] (essentially a boosted tree based ranker) to train the ranker. We used 10 fold cross validation in the training to avoid overfitting the training data.

The trained ranker is a function that takes a feature vector representing a snippet and outputs a real-valued score. Snippets are sorted according to this score.

0.5 Renaming Variables

Snippets from the web rarely fit in a user's programming context. For one thing, the retrieved snippets are unlikely to use the proper variables from user's context. To make the snippet work, the user will have to rename variables manually. Bing Code Searchautomates this process through variable renaming.

For instance, the query *"how to use the regular expression x to match the string y"* references two string variables `x` and `y` in the program context; and we may retrieve a snippet shown in Listing 1. What the user called `x` and `y` in the query is named `ptn` and `msg` in the snippet.

Listing 1: Regular expression usage pattern.

```
1  void MatchPattern(string msg) {
2    string ptn = "[a-z]+";
3    Regex strRegEx = new Regex(ptn);
4    if(regex.IsMatch(msg)) { }
5  }
```

Variable renaming replaces some snippet variables by the ones referenced in the user query. In this context, we generalize the notion of snippet variables to also include snippet constants and parameters. This is useful in cases like a slight modification of Listing 1 where a variable that simply holds a constant (namely `ptn`) is not defined and the constant is directly used. Let V_q denote the set of variables referenced in the query, for example $\{\texttt{x}, \texttt{y}\}$ (those variables can be identified from the user's context). Let V_s denote the set of variables mentioned in the snippet, for example, $\{\texttt{ptn}, \texttt{msg}\}$. The renaming finds a mapping $m = \{(v, u)|v \in V_q, u \in V_s\}$ with the restriction that a query variable is only mapped to a snippet variable if they have the same type. In other words, the variable renaming starts from all type-wise correct mappings and then ranks those mappings.

Like in the snippet re-ranking case, we follow a machine learning approach to learn how to select the correct variable mapping. The first step is to formalize the properties of a mapping into a set of features. The designed features are listed in Table 2; they capture a variable mapping in conceptual aspect, dataflow aspect and positional aspect.

Table 2: Features in variable renaming ranking

Feature name	Type	Description
Concept	real	indicates how similar the concepts of two binded variables are
Def-use	real	percentage of binded variable pairs that obey def-use rules
Method argument	real	percentage of arguments from the containing method that are binded
Return value	binary	1 if the return value in snippet is binded or there is no return value
Loop variable	binary	1 if a loop variable is binded
Fill all arguments	binary	1 if arguments of a multiple argument method call are either all binded or not binded at all
Appearing order	binary	1 if the binded snippet variables appear in the same order as query variables in user's code

0.5.1 Conceptual feature

The *Concept* feature estimates whether variables in a query and in a snippet denote the same "concept". For example, in the query *"how to use the regular expression x to match the string y"*, `x` denotes a *regular expression*; as does `ptn` in the retrieved snippet of Listing 1. This gives a clear signal that these variables should be paired. We cannot precisely capture concepts such as "regular expression" semantically, but *we can approximately decide whether two variables denote the same concept*: we associate a list of keywords to each variable from the query and from the snippet. Both lists will, in our example, contain occurrences of "regular" and "expression", which means that variable mappings that rename `ptn` to `x` should be ranked highly. We next describe how to compute the set of keywords in the query, in the snippet, and how we measure similarity.

We use the natural language processing tool OpenNLP [18] to parse a user query, and associate to each variable name the list of words that appear, in the parse tree, at a similar semantic level. For instance the parse tree of our example query is shown in Figure 3: variable *"x"* appears in a noun phrase (NP) where we also find the set of words {"the", "regular", "expression"}. Therefore we approximate the concept underlying variable `x` by this set.

For variable names occurring in snippets, we compute the set of concept keywords from the API documentation. More specifically we focus on the documentation describing method arguments: we inspect all places where the variable appears in a method call; and collect the (pre-indexed) set of words that document the associated parameter (We indexed all available documents for popular libraries. When no document was available for some library or the snippet variable was not used in any method call, we set the *Concept* feature to 0). For instance in Listing 1, variable `ptn` is used as the first parameter of the `Regex` constructor. The document for that parameter gives us the set {the, regular, expression, pattern, to, match}.

The similarity value for a variable binding pair (v, u) is the text cosine similarity between the concept of u and v, which is a value between 0 and 1 with a larger value indicating better similarity. The value of the *Concept* feature is the average similarity over all variable binding pairs (a mapping may contain more than one pair in case there are more than one variable referenced in the user query).

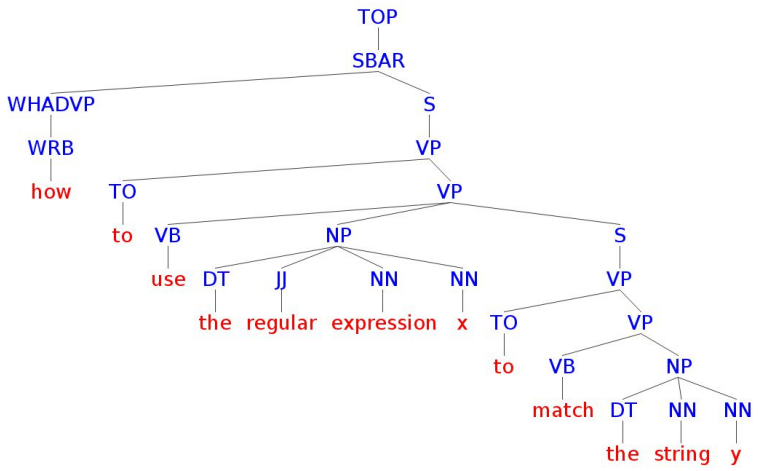

Figure 3: Parse tree for a user query

0.5.2 Dataflow features

The *Def-use*, *Method argument*, *Return value* and *Loop variable* features are dataflow-oriented. They model the dataflow that usually happen to fit the snippets into the user's context.

The *Def-use* feature is based on the following intuition (1) variables that are initialized in the user's context are more likely to be used (and less likely to be initialized) in the suggested snippet. (2) Variables that are initialized in a snippet are more likely to be used (and less likely to be initialized) in the user's context. (We exclude some trivial variable initializations to constants.)

We argue that the intuition makes sense because if the user mentions a variable in the query and the variable is already initialized, it is more likely to serve the purpose of passing information to the desirable snippet, and less likely to receive information from the snippet. Similarly, if an uninitialized variable is mentioned in a snippet, it is more likely to receive information from the snippet. and the suggested snippet. The variable `pth` is initialized in the user's context. It is less likely to be matched with the variable `line` because `line` is initialized in the code snippet. The value of the feature *Def-use* is set to the percentage of variable pairs that obey this def-use rule.

The *Method argument* favors variable mappings which bind the arguments of the snippets. If a snippet is a method declaration and this method has arguments, it is very likely that the arguments should be binded. Suppose for the query *"read file pth line by line"* in Figure 1, one of the candidate snippet is in List 2:

Listing 2: Snippet in form of method declaration

```
1  void ReadTextFile(string fName) {
2    using (StreamReader r = new StreamReader(fName)) {
3      string line;
4      while ((line = r.ReadLine()) != null) { }
```

```
5     }
6   }
```

Since the whole snippet is a method declaration and the method has one argument `fName`. It is very likely the argument `fName` should be binded to the query variable `pth`. The value of *Method argument* feature is the percentage of the method declaration arguments that are binded to query variables. If there is no such argument, the value is set to 100%.

Similar to the *Method argument* feature, if a snippet has a return value, that value is more likely to be binded to an uninitialized query variable to indicate the receiving of some information from the snippet. The *Return value* features captures this factor by setting the value to 1 if the return value is binded.

Loop variables are less likely to be binded to query variables.

0.5.3 Positional features

Positional features include the *Fill all argument* and the *Appearing order* feature. They apply when there are more than one variables mentioned in the user query.

The *Fill all argument* favors mappings that binds all arguments in a method call or bind none of them. If a method called in the suggested snippet requires two arguments and the mapping m already renames one of the arguments into a query variable, it is preferable that the other argument is also renamed by the mapping. For example, for the user query *"add variable x and value y into Windows registry key"* (`x` and `y` are strings) and the snippet in Listing 3. Since the `SetValue` method has two actual arguments `"Name"`, and `"Isabella"`, it is more likely to bind both of them with the query variables `x` and `y`. The value of this feature is 1 if a mapping obeys this rule and 0 otherwise.

Listing 3: Adding a Windows registry entry

```
1    RegistryKey key;
2    key = Registry.CurrentUser.CreateSubKey("key");
3    key.SetValue("Name", "Isabella");
4    key.Close();
```

The *Appearing order* feature favors a mapping if the appearing order of the binded variables in the snippet is the same as the appearing order of query variables in user's code. The appearance order of variables in snippets usually coincides with the appearance order of the variables in user's context. Continuing with the above example in Listing 3, if `x` appears before `y` in user's context, it is more likely that the snippet variable that `x` is mapped to appears before the snippet variable that `y` is mapped to. According to this criterion, the mapping $\{$`x` $\mapsto$ `"Name"`, `y` $\mapsto$ `"Isabella"`$\}$ is preferable to the mapping $\{$`x` $\mapsto$ `"Isabella"`, `y` $\mapsto$ `"Name"`$\}$. The feature is set to 1 if the ordering is preserved, 0 otherwise.

0.5.4 Training the variable renaming ranker

Ideally, we need to build a ranker to rank all possible mappings. But the situation is different from that of snippet ranking because in variable renaming, we are only interested in one — the correct mapping — out of many. All the other mappings are

considered wrong. The ordering among the wrong mappings is not meaningful. This suggests a binary classifier as an implementation choice: the classifier decides if a mapping is correct or wrong.

Bing Code Searchuses the Ranking SVM [7] algorithm for training a binary classifier. The basic idea of Ranking SVM is to formalize the ranking problem into a binary classification problem and use support vector machines to solve the classification problem. Ranking SVM is supervised learning.

To build the training data, we continued to use the 150 sampled queries from Bing's query log. For each snippet annotated as relevant, we also manually wrote down the correct variable mappings. This resulted in 672 mappings. These 672 mappings are positive instances. All the other type-wise correct mappings are considered as negative instances.

Note that the number of correct mappings is significantly lower than the number of wrong mappings. To create a balanced training set in which the number of correct and wrong mapping instances are equal, we generate instances in the following way: for a pair of correct mapping M_c and wrong mapping M_w, let F_c and F_w be the feature vectors calculated for them. Generate two instances, the first is a positive instance with value $F_c - F_w$ and label 1, and the second a negative instance with $F_w - F_c$ and label -1. The Ranking SVM is trained to perform the classification. During training, we used 10 fold cross validation in the training to avoid overfitting.

For a mapping, the classifier outputs a score in range [-1, 1]. The larger the score, the more likely the mapping is considered correct by the classifier. Bing Code Searchuses this score to sort the mapping candidates and takes the top one mapping.

0.6 Collecting user feedback

What we have described so far allows us to propose to the user a small selection of snippets, ranked according to their estimated relevance, and adapted to the user's context through variable renaming. These suggestions are presented to the user within Visual Studio as shown in Figure 1. From there, the user will typically select one snippet and insert it into the program text.

To provide assistance in this process, Bing Code Search records the number of times each snippet is selected by users. If a similar query is asked later, Bing Code Search will adjust the ranking by boosting the snippets that have more frequently been selected in the past.

With recorded user selection, Bing Code Search adjusts the suggestion list when similar queries are asked. It turns out that calculating query similarity is difficult, because we ideally want to capture semantic similarity rather than syntactical similarity. Our tool adopts the notion of query similarity widely used in search engines: two queries are considered similar if the returned documents are similar. In other words, Bing Code Search does not measure query similarity directly, instead, it assumes that if the resulting snippet suggestions are similar, the user queries are similar.

Since Bing Code Search remembers how many times a snippet has been selected by the users, it boosts the most frequently selected snippet to the top of the list.

0.7 Framework Implementation

Bing Code Search is implemented in a client-server architecture. The client is the Visual Studio plugin. The plugin sends user queries to the server, receives snippet suggestions from the server, records user actions and sends those actions to the server. We use three sets of servers located in three geographical areas: US, Europe, Asia. Depending on the user location, queries are automatically routed to the nearest location

The server invokes the Bing search engine to get relevant web pages, performs snippet extraction and ranking, performs variable renaming and sends the suggested snippets to the plugin. The server also stores all pre-extracted snippets from popular C# programming websites to reduce response time. The average response time per query is 1.5 second, which provides a smooth user experience.

0.8 Experimental Evaluation

We designed two experiments to evaluate Bing Code Search. The first experiment evaluates whether it suggests relevant snippets on new queries, i.e. in a setting where we don't have a history of previous queries and users, usually referred to as *cold start*. The second experiment evaluates if using Bing Code Search can save users' time compared to the de facto method of only using web search.

0.8.1 Experiment I

To evaluate if Bing Code Search can effectively answer new queries, we devised a benchmark with 150 queries. The queries are proportionally randomly sampled from a month's Bing query log according to their occurrence frequency. Some example queries are *"open file"*, *"generate MD5 hash code"*, *"upload file to ftp"*, *"serialize object to xml"* and *"check whether serial port is open"*.

For each query, we asked a programmer to construct a context in which the query can be asked. The context consists of all the code surrounding the query, such as variable declarations and definitions. The programmers were allowed to optionally change the query text, and reference contextual variables in the queries. This resulted in 80 queries with one variable, 29 queries with two variables and 41 queries with no variables. The popular occurrence of variables in queries indicates that referencing variables in queries is natural for programmers and makes it easier to express the task.

For example, for the query *"split string"*, the context devised by a programmer is shown in Listing 4.

Listing 4: Splitting string

```
1  static void Method(string[] args) {
2    string st = "dg␣xd␣we␣wx";
3    ///split string st by space
4  }
```

For each query, we retrieved 20 web pages using Bing and extracted all snippets from those web pages. We asked the programmers to annotate each snippet as relevant or

not according to the query. A snippet is annotated as relevant if it contains all the statements needed to solve the task, even if they additionally include irrelevant statements. This resulted in 3947 annotated snippets. For each relevant snippet, if the query references any variable, we also asked the programmers to write down the correct variable renaming mapping, resulting in 672 mappings. Bing Code Search is evaluated against this benchmark.

Suggestion quality

In information retrieval system evaluation, two metrics, Mean Reciprocal Rank (MRR) and Normalized Discounted Cumulative Gain ($NDCG$), are most widely used. We use them to evaluate the quality of the suggestion list given by our trained snippet ranker, compared to the original ranking from the Bing search engine.

The snippet ranker is trained using the FastRank algorithm. A 10-fold cross validation is used to avoid overfitting the training data. We repeated the training for 100 times to minimize the variance of the result due to randomness in cross validation. The numbers reported below are averaged over these 100 iterations. Variable renaming evaluation used the same process.

We compare the snippet ranking from our trained ranker against the ranking from Bing. The snippet ranking from Bing is estimated by putting all snippets from the first web page (in their appearance order) on top of the second web page, and so on. This simulates how humans browse through the retrieved web pages.

Mean reciprocal rank. The reciprocal rank (RR) for a query is calculated as the inverse of the rank of the first solution snippet. For example, if the top ranked snippet is solution, then the RR value is $\frac{1}{1} = 1$, if the second ranked snippet is solution, the RR value is $\frac{1}{2}$. The larger the MRR value, the better the suggestion list because it reveals that first relevant snippet appears earlier in the ranking. The mean reciprocal rank is the RR value averaged across all the queries.

The MRR value for the original Bing ranking is 0.70. The MRR value for the re-ranked list is 0.74, which is a 5.7% improvement. This shows that in the re-ranked list, a relevant snippet is more likely to appear earlier.

Normalized discounted cumulative gain. For a query, there are usually multiple snippets that can serve as a solution, and a user may prefer other solutions over the first suggested solution. Solution snippets are more useful when appearing earlier in the result list. The discounted cumulative gain (DCG) metric quantifies this notion. For a ranked list R, the discounted cumulative gain DCG_R is defined as:

$$DCG_R = \sum_{i=1}^{|R|} \frac{2^{rel_i} - 1}{log_2(i+1)} \tag{5}$$

where i iterates over all the indexes in the ranking; rel_i is a binary value with 1 indicating that the snippet at position i is a solution and 0 otherwise. $|R|$ gives the number of items in the ranked list. Intuitively, DCG calculates the weighted sum of the relevancy (either 1 or 0) of all the proposed snippets. The weight for each snippet degrades as we move down the list. Unlike the reciprocal rank metric which only focuses on the rank of the first solution snippet, the discounted cumulative gain DCG metric measures the

Table 3: NDCG comparison

NDCG	Original list	Re-ranked list	Improvement
NDCG@1	0.607	0.635	4.6%
NDCG@2	0.557	0.622	11.7%
NDCG@3	0.545	0.627	15.0%
NDCG@4	0.553	0.631	14.1%
NDCG@5	0.554	0.636	14.8%
NDCG@6	0.562	0.645	14.8%
NDCG@7	0.575	0.661	15.0%
NDCG@8	0.580	0.673	16.0%
NDCG@9	0.585	0.682	16.6%
NDCG@10	0.596	0.690	15.8%

quality of a whole ranking. The normalized discounted cumulative gain (NDCG) value is calculated as

$$NDCG_R = \frac{DCG_R}{IDCG_R} \tag{6}$$

where $IDCG_R$ is the discounted cumulative gain value for the *ideal ranking* for snippets of the same query, which can be obtained from our annotations in the benchmark by putting all solution snippets before the non-solution ones. $NDCG$ value has range $[0, 1]$ with a larger value indicating a better ranking.

The NDCG values at rank from 1 to 10 for both the original and the re-rank lists are reported in Table 3. The table reveals that the trained snippet ranker outperforms Bings ranking in the case of suggesting relevant snippets at all ranks. Specifically, the improvement is 4.6% at rank 1; and 15.8% at rank 10. This improvement is very significant in search engine evaluation.

Bing Code Search versus code search engines

Besides using a search engine, we also used a code search engine Ohloh [16] to search for solutions to the benchmark queries. It produced poor results: for 60 out of the 150 benchmark queries, the top 10 returned snippets did not contain any solution. This is because code search engines expect the search keywords to be program language tokens, instead of natural language words. However, users usually do not know the right tokens; this makes the code search engines much less usable, as showed by our experiment.

Variable renaming

Variable renaming replaces some variables mentioned in snippets by variables referenced in a query. In the benchmark, we only consider variable renaming for the snippets that are annotated as solutions. Some snippets need no renaming if no variable is referenced in the query; others need one or more renaming.

Out of the 672 manually annotated variable renaming mappings, 71% were correctly selected as the top from among other competing mappings (whose average number was 10.4) for the snippet and context in question.

0.8.2 Experiment II

To find out if programmers can save time in code search and reuse with Bing Code Search, we performed a baseline study involving 14 programmers: 7 programmers were asked to use only the web search to complete 45 queries, and another 7 programmers to complete the queries using Bing Code Search. Those 45 queries are selected from the benchmark and they cover different API usages, ranging from simple ones such as *"download file from url"* to more complicated ones such as *"upload to ftp"*.

Table 4: Time performance Comparison

Action	T_{WEB}	$T_{Bing\ Code\ Search}$
Snippet selection	69.9	54.2
Snippet adaptation	41.1	25.5
Total	111.0	79.7

Table 4 presents performance data (time in seconds) for this experiment. Column T_{WEB} reports the average time in seconds that a programmer spent in snippet selection and adaptation by using only the search engine. Column $T_{BingCodeSearch}$ reports programmers' average performance by using Bing Code Search. The numbers show that, Bing Code Search helps programmers save 28% of the time.

0.9 Lessons Learned

The experimental evaluation helped us to demonstrate the effectiveness of the proposed framework. Shipping the extension to a large public and collecting usage data allowed us to understand how programmers are using our technology. This section shares our findings. We believe that many of them are general and apply to the problem of code search in an IDE. The usage data were recorded during three weeks after the release. Queries were collected from around 20,000 users, who issued an average of 3,000 queries per days.

0.9.1 Ease of usage and user excitement

The first finding is that users immediately understand the benefit of the extension and how to use it. Few users asked us about usage instructions. A simple feature is one of our design goals and it is very important to grow a large user community. Further, users are excited about this advanced code search and reuse feature. Thousands of users shared this extension through social networks; many left very positive feedback: "Awesome Idea. Its really helpful feature for all coders", "Amazing. Love it. For those moments where I forget", "Not an easy job, but surprisingly neat, and correct!".

Table 5: Solving complex tasks

User	Time	Query
527	2/20 2:24PM	read excel file
527	2/20 2:26PM	regex numbers only

Table 6: High-level auto-completion

User	Time	Query
67	3/4	open database connection
67	3/5	open database connection
67	3/12	open database connection

0.9.2 Habit forming

The second finding is that there are already recurring users (users who issued queries on more than one day). From recorded telemetries, we observed that 48% of the users issued queries on one day; 21% on two days and 31% on three days and above. Further, we can also see that some developers already start to form search habits. A usage habit means that a developer uses the feature again and again. This is a good indication that the feature offers continuous value. We observed two different habits: solving complex tasks and high-level auto-completion.

A complex task requires multiple steps, and we observed some users issued different but related queries in a short period of time, indicating they used the tool to guide them through the steps. Table 5 shows such an example: user 527 issued two queries in 2 minutes. The first solves the task of reading data from Excel file; and the second query solves the task of parsing numbers using regular rexpression. This is a typical pattern for processing spreadsheets. The fact that the user issued these related query in a short time period shows that she is in the process of solving a multi-step task.

Bing Code Search enables high-level auto-completion: auto-complete not only a single method, but also a complete solution for a task. We observed that some users issued the same queries on different days, indicating that the users know how to solve the task, and they rely on the tool for a faster auto-completion. Table 6 shows a concrete example from user 67. This user issued the same query on three different days, with other queries in between, showing that she practically used the tool to auto-complete a task.

0.9.3 Criticisms

We also received negative feedback. Many developers complained about the absence of a keyboard shortcut to trigger the Bing Code Search window, see Figure 2). Many complained about the lack of compatibility with older versions of Visual Studio. Many more complained about the lack of support for their language, e.g., C++, or Visual Basic.

0.10 Conclusion

Programmers encounter difficulties when using APIs. This paper describes the Bing Code Search framework that suggests code snippets from natural language queries. Bing Code Search uses a search engine to retrieve web pages, extracts code snippets from these pages and re-ranks those snippets to achieve better precision. It then adapts the snippets into the user's context through variable renaming. The framework also collects user feedback to improve future suggestion quality.

Two experiments are used to evaluate the framework. They show that Bing Code Search is able to suggest better results than the Bing search engine, and that it helps programmers save time in API related programming tasks.

Telemetries collected from real users revealed that Bing Code Search is easy to use and some of them have started to form usage habits. This shows that Bing Code Search can provide continuous value for developers, instead of just one cool tool which will be forgotten after a few trials.

For the moment, Bing Code Search only suggests C# snippets. The basic ideas are independent of the language, the IDE and the underlying search engine and our findings should apply to other settings.

After three weeks, this free extension has been downloaded by more than 20,000 users who issue on average 3,000 queries per day. We believe that Bing Code Search is the most widely used tool in its category. This demand has prompted efforts to integrate other programming languages including C++, JavaScript, VB.Net and F#. We also received requests from programming websites to be registered as a source of information for Bing Code Search, including GitHub, CodePlex and CodeProject.

We provided an online version of Bing Code Search at **`http://codesnippet.research.microsoft.com`**. A video from the same page shows the extension in use.

Bibliography

[1] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. CHI, pages 513–522, 2010.

[2] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.

[3] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for java using free-form queries. In *FASE*, pages 385–400. 2009.

[4] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.

[5] http://code.google.com/codesearch.

[6] S. Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, pages 317–330, 2011.

[7] R. Herbrich, T. Graepel, and K. Obermayer. *Large margin rank boundaries for ordinal regression*. MIT Press, Cambridge, MA, 2000.

[8] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[9] V. Le, S. Gulwani, and Z. Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys*, 2013.

[10] http://en.wikipedia.org/wiki/Learning˙to˙rank.

[11] G. Little and R. Miller. Keyword programming in Java. *ASE*, pages 37–71, 2009.

[12] D. Lorenzoli, L. Mariani, and M. Pezze. Automatic generation of software behavioral models. *ICSE*, pages 501–510, 2008.

[13] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. PLDI, pages 48–61, 2005.

[14] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. OOPSLA, 2012.

[15] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *ICSE*, pages 69–79, 2012.

[16] http://code.ohloh.net.

[17] S. Oney and J. Brandt. Codelets: linking interactive documentation and example code in the editor. CHI, pages 2697–2706, 2012.

[18] http://opennlp.apache.org.

[19] S. P. Reiss. Semantics-based code search. ICSE, pages 243–253, 2009.

[20] http://www.microsoft.com/en-us/download/details.aspx?id=27746.

[21] N. Sahavechaphan and K. Claypool. XSnippet: mining for sample code. *SIGPLAN Not.*, pages 413–430, 2006.

[22] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz. Sando: An extensible local code search framework. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 15. ACM, 2012.

[23] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. *SIGPLAN Not.*, pages 281–294, 2005.

[24] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[25] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. ASE, pages 204–213, 2007.

[26] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 319–328. IEEE Press, 2013.

[27] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. ASE, pages 295–306, 2009.

[28] D. Wightman, Z. Ye, J. Brandt, and R. Vertegaal. Snipmatch: Using source code context to enhance snippet retrieval and parameterization. UIST, 2012.

[29] J. Yang and L. Tan. Inferring semantically related words from software context. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 161–170. IEEE, 2012.

[30] K. Yessenov, Z. Xu, and A. Solar-Lezama. Data-driven synthesis for object-oriented frameworks. *SIGPLAN Not.*, pages 65–82, 2011.