PrefFinder: Getting the Right Preference in Configurable Software Systems

Dongpu Jin
Dept. of Comp. Sci. & Eng.
University of Nebraska-Lincoln
Lincoln, NE 68588, USA
djin@cse.unl.edu

Myra B. Cohen
Dept. of Comp. Sci.& Eng.
University of Nebraska-Lincoln
Lincoln, NE 68588, USA
myra@cse.unl.edu

Xiao Qu
Industrial Software Systems
ABB Corporate Research
Raleigh, NC, 27606, USA
xiao.qu@us.abb.com

Brian Robinson
ABB Inc.
Raleigh, NC, 27606, USA
brian.p.robinson@us.abb.com

ABSTRACT

Highly configurable software, such as web browsers, databases or office applications, have a large number of preferences that the user can customize, but documentation of them may be scarce or distributed. A user, tester or service technician may have to search through hundreds or thousands of choices in multiple documents when trying to identify which preference will modify a particular system behavior. In this paper we present PrefFinder, a natural language framework that finds (and changes) user preferences. It is tied into an application's preference system and static documentation. We have instantiated PrefFinder as a plugin on two open source applications, and as a stand-alone GUI for an industrial application. PrefFinder finds the correct answer between 76-96% of the time on more than 175 queries. When compared to asking questions on a help forum or through the company's service center, we can potentially save days or even weeks of time.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Configurable Systems; Testing; Debugging

1. INTRODUCTION

Many software systems today are highly configurable. Users can customize the program's behavior by choosing settings for a large number of preferences. Preferences control which features are used (or excluded) during the program execution, and most systems support the selection of these at both compile time and runtime. During development, testing, maintenance and when providing end-user technical support, engineers need to manipulate a system's preferences to mimic user behavior and ensure that correct execution occurs under a wide range of user profiles.

Most large real systems provide multiple ways to access and modify preferences [14]. On the user interface there may be a preference menu that is easily accessible, containing a core set of preferences. For advanced users who want to manipulate the less common options, or who need to automate the configuration process, changing preferences can be achieved by modifying values in preference files, or through interaction with a runtime API which connects to the active *preference database* [14].

Despite the flexibility and availability of configuring and manipulating how a program runs, it is often non-trivial to determine which preference is tied to a specific behavior (or to a specific element of code). For instance, if a developer knows that a preference in the Firefox browser found on a menu is called *Always show the tab bar*, he or she may not be able to quickly determine what the real preference name is in the preference database or files (i.e. *browser.tabs.autoHide*). And if a user wants to change the tab bar behavior, they may spend a long time searching through menus to find out where such an option can be modified. In recent work, Wiklund et al. reported that the majority of the impediments for novice testers were in configuring the testing tools to use the correct parameters and environment [16].

Rabkin and Katz highlight the lack of documentation that exists for preferences, including knowing the valid value domains for each of the preference options [30]. Making this worse, we have observed in recent work [14] that the locations for manipulating configurations within a system are often distributed and only a limited number can be manipulated by the menu (e.g. only 126 of 1957 Firefox preferences are accessible from the menu). We see similar trends in industrial systems, such as those studied at ABB. To address this issue, Rabkin and Katz developed a static analysis technique that reverse engineers the configuration options from code either for configuring systems or for diagnosing errors [29, 30]. Zhang and Ernst have developed another analysis to identify which configuration option causes a fail-

ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2643009.

ure [44] or has caused the system behavior to change in an undesirable way due to evolution [45], but this is limited to situations where the differing behaviors are known and can be demonstrated.

Both of these approaches identify preferences at the source code level, but it is non-trivial to map them back to the preference database names and/or to the menu items [14]. Furthermore, the use of static analysis means that these techniques are programming language dependent and many of the highly configurable systems like Firefox are written in multiple programming languages with preference code distributed throughout [14].

Some programs provide built-in search utilities tied into their documentation (as in an ABB system), or to the runtime preference database (as in Firefox), but these primarily use keyword searches forcing the user to know exactly what they are looking for. Consider a preference found in Firefox, *browser.download.DownloadDir*. It determines the default download directory when users save a file. There is no menu setting for this option, but Firefox provides a utility, `about:config`, to look for this. If the user happens to search using the keyword *download*, they will find this option and can then modify its settings. If instead, they search for *directory*, they will not (because the keyword is *dir*). Browsing through all preferences in `about:config` is not useful with over 1900 in the current versions. And if the user is working on a system like LibreOffice, they have a hierarchical directory to search that has over 30,000 choices [14]. Instead, there is the need for a more natural way to interact and find preferences in highly configurable systems.

In this paper, we present a natural language processing (NLP) based framework called *PrefFinder*. In PrefFinder, a query is an input in natural language. PrefFinder first parses both the preferences and the user query, informed by dictionaries and lexical databases. The queries and preferences are then matched, ranked and returned to the user. We can tie this into the runtime APIs of the applications to provide descriptions of the returned preferences and to allow directly modification of the chosen preference. PrefFinder has been instantiated on two open source and one industrial application. In a case study on more than 175 real queries from users and developers across these three systems, we show that PrefFinder is effective and has potential to save time over existing techniques.

The contributions of this work are:

1. PrefFinder: An extensible framework to provide natural language interactive querying of preferences
2. An implementation for three different applications
3. A case study on more than 175 queries demonstrating its potential usefulness

The rest of this paper is laid out as follows. In the next section, we provide a motivating example. We then present PrefFinder in Section 3. In Sections 4 and 5 we evaluate and show the results of our case study. In Section 6 we describe related work. We end with our conclusions and discuss future work in Section 7.

2. MOTIVATION

We motivate our research with two open source applications, Firefox and LibreOffice using data from [14]. Firefox is a web browser that contains approximately 10 million lines of code. Like many modern applications, it is developed using various programming languages including C++(41%), C (21%), JavaScript (16%), Java (3.1%), and Assembly(1.2%). In addition it uses XUL, a markup language for interface functionality and this is where some preference options are manipulated. It is highly configurable with 1957 preferences in the cited version (Ubuntu, Firefox version 27.0a1). In addition, the browser can be extended by installing third party extensions, which can introduce an arbitrary number of additional preferences. LibreOffice [6] is an office productivity suite of tools that includes a word processing, spreadsheet and presentation module. It is written in C++ (82%), Java (6%) and uses additional scripting languages such as python. The number of preferences associated just with the Writer application is 656. There are similar numbers of preference for each of the other applications in addition to ones common to all applications [14].

Since both of these applications are written in multiple languages this presents challenges for analyses targeted at one language, such as Java (i.e. [29, 30, 44, 45]). In systems like these, there are two common approaches for customizing preferences. One is to make changes via an option (or preference) menu, by clicking buttons, selecting check boxes, and checking radio buttons, etc. This approach is intuitive to use for novice users, but as mentioned, the option menu only contains the most commonly used set of preferences and it is only a small subset. In Firefox, if one wants the tabs to appear above the URL bar rather than below, there is an option *browser.tabs.onTop* which can be set to true. But this option is not available in the preferences menu. In fact, the removal of this from the user menu is the subject of a post and user complaint on the Firefox forum.[1]

The second approach is to modify preferences directly via the preference files and/or a preference utility. Firefox maintains its preferences for persistence via a set of preference files at various locations (and during runtime via a hash table in memory). At startup it reads these files and stores the contents in memory. Firefox has by default 11 such files [14]. LibreOffice maintains a directory of preferences for external storage that it reads at startup as well (over 190 of them). In LibreOffice, for instance, there are many directories such as *soffice.cfg* with files such as *menubar.xml* that can be modified to determine which buttons appear on which toolbar.

Both applications provide runtime APIs which can examine the internal/available preferences [23, 39]. Firefox's `about:config` utility uses a regular expression keyword search, but this requires an exact match. For instance, if the user types *tabs* or *tab* for the previous preference they will find this option. However, if they type *browsers* or *tabbing* they will not find it. LibreOffice does not have an about:config utility. It has a file called *registrymodifications.xcu* which is difficult to parse and understand. Most users of LibreOffice will probably use menu options to change their preferences which may not scale for larger maintenance tasks.

In each of these scenarios, the preferences are represented as mid-level variable names (i.e. they are neither code nor natural language) and there is no direct traceability between the options menu and these names or down to the code. In the work of Rabkin and Katz [29, 30] and Zhang and Ernst [44, 45] preferences names are found and returned

[1]`https://support.mozilla.org/en-US/questions/991043`

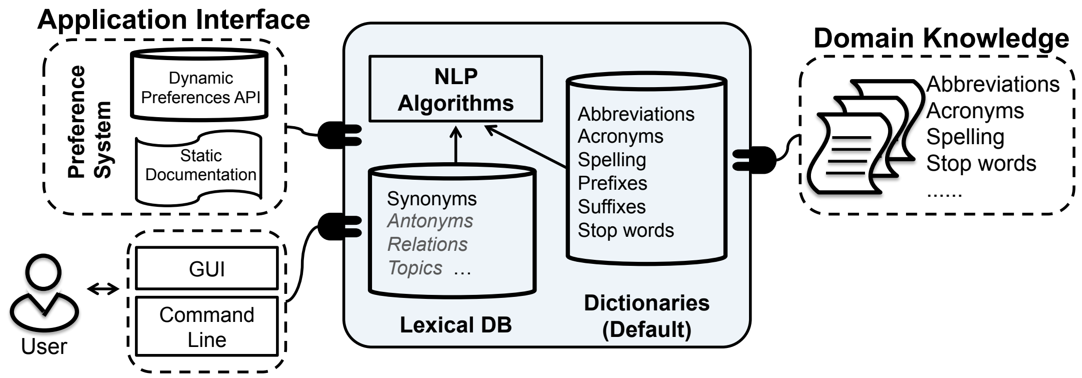


Figure 1: PrefFinder framework architecture

from the source code. But these may or may not match the names found in the intermediate preference files where the users manipulate preferences directly [14]. Therefore, while useful (and a possible complement to our system), they will not provide the information we seek. Before one can automate any sort of configuration manipulation, or allow technical support to help users change a configuration, they must first translate from the intended behavior to these preference names and then find a way to modify them.

Finally, configuration spaces are rapidly changing. We need to be able to easily re-evaluate questions when a version is updated. For instance, the example discussed in the introduction, and the work of Zhang and Ernst [45] highlight the changing configuration space and the need to find options over and over again. In the next section we present PrefFinder which will overcome most of these challenges.

3. PREFFINDER

Figure 1 shows an overview of the PrefFinder framework. On the left of this figure we see a (customizable) *Application Interface* where interaction with the user and the application itself takes place. On the right we see *Domain Knowledge* that allows information to be added to our databases. In the center lies the core component of PrefFinder. This contains natural language processing algorithms, dictionaries and lexical databases. It can be used on its own or packaged into a plugin or GUI application or run at the command-line which permits automation of multiple queries at a time.

In PrefFinder, the users enter a short description or question (in English) about what features or functionality of the system they want to lookup (customize). They also specify a number of results to display, and PrefFinder will return a list of ranked preferences (with a value showing the score) and a description of each result if available. Users may enter arbitrary English sentences with different punctuation, numbers, mixed-case letters, and may use different forms of the language such as present participle (e.g., *closing*) and plural (e.g., *tabs*).

To instantiate the framework, *preference extraction* is first performed to build the preference system. Next, the core component runs several steps to identify the correct preferences that best match the query: *splitting*, *parsing*, *matching*, and *ranking*. Finally, the *documentation* step connects preference descriptions to corresponding preferences returned in the results. Optionally, for instantiations that use a dynamic API to extract the preference system, PrefFinder is able to also *update* the preference via its GUI. Let us take Firefox as a running example. Suppose the user types a query "*Firefox 17.0 doesn't warn me when closing multiple tabs any more.*". As reported online, there are four different preferences that a user can set to modify this: *browser.tabs.warnOnCloseOtherTabs*, *browser.tabs.warnOnClose*, *browser.showQuitWarning*, and *browser.warnOnQuit*. PrefFinder is expected to find at least one of them.

3.1 Preference Extraction

The first step is preference extraction from the application. To extract preferences, PrefFinder can utilize different types of information, such as a static analysis like that of Rabkin and Katz [30] or it can also use static artifacts such as a user manual. We can also use APIs to extract the preferences dynamically. In our prior work we used the runtime APIs to automatically extract preferences from two running applications' databases [14]. For one of the systems, there are APIs that directly return a set of all the preferences. For the other system, we traversed the hierarchical structure in a depth first search fashion. We can also combine static and dynamic methods to get a more complete view of the preferences. In the running example we would extract the preferences from Firefox using the dynamic API and obtain approximately 1900 preferences in our list.

3.2 Splitting

Once we have the preferences, we normalize them into bags of words. PrefFinder begins by taking in the the system preferences and parsing/splitting them into sets of keywords (see [13] for more details of the parsing algorithms we used). Our identifier splitting algorithms are based on the work of Enslen et al. [8]. Preference names are usually represented as arbitrary strings, such as our running example from Firefox *browser.tabs.warnOnCloseOtherTabs*, or *org.eclipse.jdt.core.compiler.codegen.targetPlatform* as in Eclipse, and */org.openoffice.Office.Recovery/RecoveryList* as in LibreOffice. As in a program variable, a preference name must be a sequence of characters without any white space. *Soft words* (individual dictionary words), within a preference name are separated by *word markers* such as a period(.), underscore (_), dash (-), backslash (/), or camel case letters [4,9]. After splitting the words, the remaining identifiers are called *hard words*. Once the initial separation via word markers is complete, we next use a camel case splitting algorithm. We found that this often does not provide an accurate split (i.e. some words are still joined or are split too much), so we use an additional *same case* splitting algorithm and a backward greedy algorithm based on the work of Enslen et al. [8].

3.3 Parsing

To incorporate meaningful (code related) words during parsing, we compiled a dictionary based on the one used by Hill et al. [11], which is derived from *iSpell* [10]. It is a list of computer science abbreviations and acronyms (such as *SYS* and *URL*) [1]. We also adopt a prefix list and a suffix list from the work of Enslen et al. [8] to identify commonly used prefixes and suffixes (such as *uni-* and *-ibility*). Our dictionaries are available online (see Section 4).

Once the preferences are split, we can parse them along with the user queries to extract a set of relevant *keywords*. Since the queries are to be run against identifier-like names, we have adopted a set of rules that limit what keywords are extracted. The parser removes words with leading numbers, special symbols and punctuation, and converts all of the letters to lowercase. After this step, the user query in our example becomes *firefox doesnt warn me when closing multiple tabs any more*.

Some words, such as *doesnt*, *me*, *when*, *any*, *more*, do not provide domain relevant information in our context. These words are commonly referred to as *stop words*. The parser filters stop words prior to further processing, using a stop words list. At this step, as shown on the back-end of Figure 1, a domain-specific dictionary can be plugged in. In our example, several words are added to the stop word list such as *firefox*, *libreoffice*, *openoffice*, *org*, and *office*, which are tied to specific applications and carry little discriminating power when it comes to configurations. After this, the above query becomes *warn closing multiple tabs* which only contains the keywords that carry the core information.

The user query has been shortened without losing the core information, however, the query may still fail to match if the user expresses the same concept using words in different forms or uses a different word with similar meaning. Preference names are often made up of *root words* (for example, *close* rather than *closing*). In another example, a user may use *shutdown* to mean *close*, something that an exact match will not find. To alleviate these issues, PrefFinder integrates WordNet [21], a lexical database for English, that converts user query and preference words to root forms and expands the keywords in a user query with their synonyms (as shown in the core component of Figure 1). In our running example, WordNet converts the word *closing* in the query to its root form *close* and expands it with synonyms such as *shutdown, shutting, closedown, closing, closure, completion.* As before, a domain-specific database can be added. Figure 1 shows additional lexical information in grey (i.e. antonyms, relations and topics) that are available in WordNet, which we will incorporate into PrefFinder in future work.

3.4 Matching and Ranking

Once we have parsed both the preferences and the query, the next step is to suggest preferences that are most relevant to the user query. To compute the similarity for each (query, preference) pair, we adopt a *Fast TF-IDF* algorithm [19], a variant of the classic information retrieval weighting scheme *term frequency-inverse document frequency* (*tf-idf*) [4,33].

A user *query* (q) contains a bag of words and each word in q is a *term* (t). Each preference name is considered a small *document* (d) that also contains a bag of words. A preference system that consists of N preferences forms a *collection* (c) of size N. *Term frequency* ($tf_{t,d}$) is defined as the number of occurrences of a term t in the document d. The value of $tf_{t,d}$ equals zero if t is not in d. *Document frequency* (df_t) is defined as the number of documents in the collection that contains the term t. The value of df_t equals zero if t does not exist in any of the documents in the collection. The *inverse document frequency* (idf_t) is defined by the equation:

$$idf_t = \log \frac{N}{df_t},$$

where df_t is the document frequency of term t and N is the number of documents in the collection. Note that if a term exists in many documents, it often carries less discriminating power (df_t is large, and thus makes idf_t small). Hence, idf_t can be used to reduce the effect of terms that appear in too many documents. The weight (or *tf-idf*) for a term in d is defined by the equation:

$$tf\text{-}idf_{t,d} = tf_{t,d} \times idf_t,$$

which is the product of the term frequency and the inverse document frequency for that item (the weight equals zero if the item only occurs in d but not q). As can be seen, a term in d would have a heavier weight if it occurs many times in a few documents (both $tf_{t,d}$ and idf_t are large). On top of the *tf-idf* weight, we impose an additional scale factor which reduces the the effect of synonyms, by scaling down their weight. Our matching favors the term that has an *exact match* in the original user query. We experimented with a series of scale factors on the Firefox preference set and found that 0.4 works best for a *synonym match*. Thus, the overall similarity score for a (query, document) pair is computed as the sum of *tf-idf* weights for all the items that occur in both the query q and the document d by the following equation:

$$\text{score}(q,d) = \sum_{t \in q} tf\text{-}idf_{t,d} \times scale,$$

where *scale* equals to 0.4 for synonyms, and 1 otherwise.

Table 1: Ranking terms in the correct preference for the example query

item in q	*tf*	*df*	*idf*	*tf-idf*	*scale*	weight
warn	1	28	0.6173	0.6173	1	0.6173
closing	1	47	0.3924	0.3924	1	0.3924
multiple	0	-	-	-	-	0
tabs	2	55	0.3241	0.6482	1	0.6482

In the *Fast TF-IDF* variant, we first build a *posting list* for each term t in the query q by collecting *relevant* preferences from the entire preference space. A posting list is a list of preferences that match t (either an exact match or a synonym match). Posting lists directly associate relevant preferences to query terms, which avoids repeated examination of the entire preference space. The length of a posting list is the number of preferences that contains t (df_t). We then calculate scores for each preference as shown in Algorithm 1. Each query term calculates its own weight (outer loop) which is added to the score of every preference in its

Algorithm 1 Pseudocode of core algorithm that calculates preference scores using *Fast TF-IDF*

```
1:  Input user query $q$, posting lists
2:  Output a collection $c'$ of preferences with score
3:  for each $t$ in $q$ do
4:      calculate $idf_t$
5:      for each $d$ in $t$'s posting list do
6:          $d$.score += $idf_t \times tf_{t,d} \times scale$
7:      end for
8:  end for
9:  $c' \leftarrow$ preferences in posting lists
10: return $c'$
```

posting list (inner loop). The algorithm returns a collection of preferences with a score.

Consider the running example, where the bag of words after parsing (without the synonyms) are {*warn, closing, multiple, tabs*} for the query q and {*browser, tabs, warn, on, close, other*} is the corresponding preference d (*browser.tabs. warnOnCloseOtherTabs*). There are total 116 preferences relevant to q ($N = 116$). Table 1 shows the statistics of each term in q (the query). The overall score is the sum of the weights of all the terms ($0.6173 + 0.3924 + 0 + 0.6482 = 1.6579$). Note that the term *close* in d is a root form of term *closing* in q, and thus is considered as an exact match with a scaling factor of 1. The term *multiple* fails to match any word in d and contributes zero weight.

After assigning each preference a similarity score for a given query, all preferences are ranked in decreasing order with respect to the score. The top n preferences (n is a parameter specified by the user via PrefFinder front-end UI) are sent to the front-end and displayed.

3.5 Documentation and Update

This part of PrefFinder is optional. We can use external documentation (such as that found on the Firefox user website) and connect this to our found preferences, providing potentially useful information for the user. In our example we would get a brief description (if available) for each found preference from the documentation written by the user community [25] [22]. In the running example there is no documentation for the first solution (so this will be null), but the second solution *browser.tabs.warnOnClose* has the following (partially eluded) text which we append in the PrefFinder result list: "Warn when closing window if more than one tab open; True(default): The browser will prompt for confirmation when closing the browser ..... this can be changed via Tools → Options → Tabs → Warn ... ". If we are connected to the preference database we can now change the value directly in memory, which will modify the external system and the preference files, and be reflected in the current state of the application. This works in the same way the `about:config` works and we have implemented this in our Firefox and LibreOffice versions of PrefFinder.

4. CASE STUDY

We perform a case study aimed at evaluating PrefFinder that asks three research questions. Supporting data on the queries used and the associated results can be found on our website.[2]

[2]`http://cse.unl.edu/~myra/artifacts/PrefFinder_2014/`

RQ1: *What effort/customization is required to instantiate PrefFinder for different systems?*

RQ2: *How effective and efficient is PrefFinder at finding the correct preference?*

RQ3: *How does PrefFinder compare with existing approaches?*

The first question is used to qualitatively evaluate the generality and applicability of our framework across different types of configurable systems. The second question evaluates the quality of PrefFinder's search mechanism using both accuracy (match success) and efficiency (time). The last question examines the current state of the art for two of our systems and evaluate these against the PrefFinder queries.

4.1 PrefFinder Versions

We built PrefFinder versions using two open source applications from different domains (Firefox and LIbreOffice) and selected an industrial application developed at ABB to avoid open source bias. All systems are large, highly configurable, and have a dedicated user base. In addition we have shown that they all have complex configuration mechanisms [14]. Basic information about each application (version number, and number of preferences extracted for use in PrefFinder) is shown in Table 2.

Table 2: Application version and preferences

Application	Version	No. of Preferences
Firefox	18.0.1	1833
LibreOffice	4.0	36,322
ABB_S	-	935

All three applications manage their preference database slightly differently and provide different configuration interfaces for the users. We describe each system next as well as the the use case of how we expect someone will interact with the given system.

Firefox Web Browser. For this study we chose Firefox version V18.0.1 running on Ubuntu 12.04.1. We note that PrefFinder itself is not operating system or version specific (and we have installed it on different versions and platforms of Firefox), but all of the results we present are from the specified version. Firefox has a large preference system that is available dynamically via an API. Although it is not the complete set of preferences [14], this is the same set of preferences that would appear in the `about:config` utility. We use the Mozilla XPCOM API [23] to extract the existing preferences at runtime. We also included additional descriptive information gathered by merging our results to the documentation written by the user community [22] [25] and allow the user to change the found preference value directly from within PrefFinder. The use case for this query is similar to the use of `about:config`.

LibreOffice Application. We used LibreOffice, version V4.0 for this study. LibreOffice has over 30,000 preferences in total [14] contained in a hierarchical database based on a specific XML schema [39]. We expect a different use case in this type of system given the complexity of the preference database and assume that the preference modification will be done via the user interface. For this version, we model a person who is trying to change the system behavior using the menu. To extract the preferences we used the API which connects to the dynamic database as detailed in the online user guide [39].

ABB$_s$ Industrial Application. Our last application is an industrial software system developed at ABB (ABB_S). All of its important user preferences can be accessed and modified in preference files. The definition and description of these preferences are available in two online documents, denoted as ABB_{D1} and ABB_{D2}. In the documentation, real preference names are used as they appear in the preference files. There are 524 preferences defined in ABB_{D1} and 411 in ABB_{D2}. The system provides a help keyword match utility (but only for preferences in ABB_{D1}) similar to `about:config` except that the user can type in multiple keywords at a time. In fact, they can type in the entire query. The keyword match will do an exact match on all entered keywords, therefore any extraneous (or incorrect) words will result in a failed search. For this use case we assume that the person doing the search is a technical support engineer who is trying to help customers who have called in on the help line. It can be time consuming to determine what the user has done (or is trying to do) to the system preference files, therefore we build this version of PrefFinder to see if it will help in this scenario.

4.2 Obtaining and Running User Queries

For each of our systems, we obtained real user queries. For Firefox and LibreOffice we went to online user forums (see [2] [24]). We searched the forums using the keywords *preference* and *configuration* and selected the first 100 (for Firefox) and first 25 for LibreOffice that had a solution (or solutions) to serve as an oracle. For the queries obtained from the Firefox forums the oracle is simple to check since in most cases the actual preference name is returned. In the LibreOffice forum this is not as straight forward. Therefore, we restricted our queries to ones in which a series of menu steps are provided (which matches our intended use case). We then performed the example steps, identified which preference changed in the preference database, and then verified this by ensuring that when we change that preference directly in the system, that particular menu item changes as well. This preference then serves as our oracle.

For ABB$_s$ we were unable to access the internal support log, so we asked several independent system engineers to write queries that they thought a user would ask based on their experience. We only briefly described PrefFinder to them without explaining the search mechanisms used in the tool. We obtained 52 queries, of which 27 queries are from the preferences in ABB_{D1} and 25 queries are from preferences in ABB_{D2}. For each query, they also provided answers which serve as our evaluation oracles.

We then used PrefFinder to run each of the queries on each system. We utilize command line versions and run the queries 10 times each to get timing information. We also compared a sample of results from the command-line and GUI versions to validate that we are reporting the correct results. Table 3 shows a few example queries from the Firefox and LibreOffice forums. We provide the full set of open source queries on our website.

4.3 Metrics

To answer RQ1, we qualitatively assess if it is feasible to build multiple versions of the system and ask what is involved in customizing each system. To answer RQ2, we run all queries and evaluate whether the correct answer is found and at which rank (for effectiveness). We also report the PrefFinder execution time averaged over 10 executions and a *break even* (BE) time for our open source systems (for efficiency), described below. The execution time is the time that a query takes to run. The evaluation time for the returned result, however is harder to measure. Once a person has the preference result list they must determine which one is correct. Since we cannot accurately predict (without bias) the time it would take someone to evaluate each preference returned by PrefFinder, we report a metric called the break even time. This metric tells us how long one would need to spend per answer in PrefFinder to make the tool no better than an existing approach (in this case a user forum). We use the oracle pages from the user forums and make the assumption that this is the first time this question was asked (i.e. the question cannot be obtained by a simple web query). Since we obtained our queries from the forums, we can examine the time stamps from when the question was initially asked and when it was first answered correctly. We then use this time as the time it takes to answer this as a new question on the forum.

To then calculate the break even time we ignore the execution runtime (but report it) since it is less than a minute in all cases. We then calculate it as: $BE = \frac{ForumTime}{Rank}$. If for instance, an answer is found in the 3rd rank position and the forum takes 30 minutes, it means that as long as a user can evaluate each result returned by PrefFinder in less than 10 minutes on average, it will perform better.

To answer RQ3, we compared the Firefox queries with the `about:config` utility and the ABB$_{D1}$ queries with its search utility. For this we had to split our queries into its constituent keywords and try each one individually. For the ABB$_{D1}$ system, since its utility can search on multiple keywords at once, we try the original query and then we try all keywords together (Key_{ALL}) followed by each one individually. The success (rate) and ranking of these results are the metrics that we compare with PrefFinder.

4.4 Threats to Validity

The first threat to validity is that of generalization. We have built PrefFinder on three different systems with different preference mechanisms and one is industrial so we believe that this is representative of many real systems. We had to select the queries from the forums, but we used a systematic approach and did not try the queries on PrefFinder before selecting them. The queries obtained from ABB may have some bias since the engineers (unlike in tech support in the real use case) are familiar with the system and the types of questions users ask. But we believe they are still representative and may in fact bias the results towards the existing approach. Finally, we may have mistakes in our data, but we have cross-validated our questions using both a manual and the command-line version. We are also making our open source data available on line.

5. RESULTS

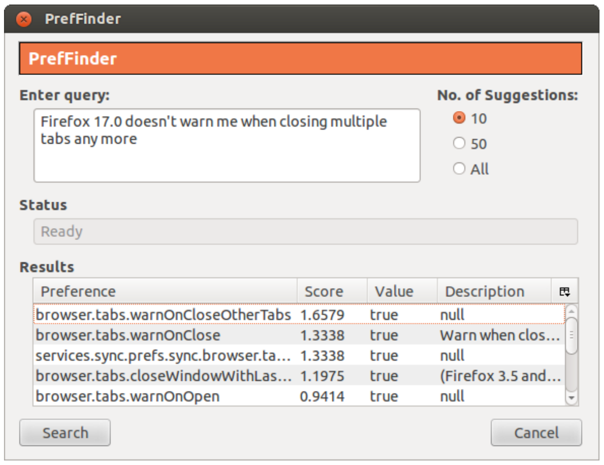
In this section, we describe our results for each question.

5.1 RQ1: Different Instances of PrefFinder

Figure 2 shows a screenshot for one of our versions of PrefFinder – a plugin for Firefox. On the screen you can see the query, the result (for the first 10 answers), including

Table 3: Sample queries from the user forums

Query	Preference Answer
FireFox	
(1) How to change permanently the Search Engine?	keyword.URL
(2) Is there an about:config entry to toggle Search Example.com for selected text automatically switching to the tab it opens?	browser.search.context.loadInBackground
LibreOffice	
(1) Do you have a way to accept or translate Excel macros?	org.openoffice.Office.Common/Security/Scripting/MacroSecurityLevel
(2) Default font color on Button in Calc	org.openoffice.Office.UI/ColorScheme/ColorSchemes/ org.openoffice.Office.UI:ColorScheme['LibreOffice']/FontColor/Color
(3) How do I disable paste via middle mouse button?	org.openoffice.Office.Common/View/Dialog/MiddleMouseButton



Enter query:

Firefox 17.0 doesn't warn me when closing multiple tabs any more

No. of Suggestions: [✓ 10] 50 All

Status

Ready

Results

Preference	Score	Value	Description
browser.tabs.warnOnCloseOtherTabs	1.6579	true	null
browser.tabs.warnOnClose	1.3338	true	Warn when clos...
services.sync.prefs.sync.browser.ta...	1.3338	true	null
browser.tabs.closeWindowWithLas...	1.1975	true	(Firefox 3.5 and...
browser.tabs.warnOnOpen	0.9414	true	null

Figure 2: PrefFinder prototype user interface

the ranking score, the current value of that preference, and a description when it is available. In this case our query is the example query used in Section 3. Both the first and second answers match our oracle (there were 4 solutions). Although we have no description for the first preference, we do for the second which will make the user's choice easier. If the user wants to change the value for this option, they can simply click on that row and it will directly modify the running application preference database or files. For Firefox, we were able to create a working version that can interface with the dynamic configurations to match our use case and provide both documentation and an update facility. Since this was our first instantiation and the easiest use case we did not encounter any problems. To customize this for Firefox (i.e. interfacing with the dynamic API and documentation) took about a day of programming time.

For LibreOffice we also implemented a plugin. We did not change the core system, but only rewrote the interface to the application. We used the dynamic API [39] to access (and modify) preferences. This took us a bit longer (approximately two days) since the documentation on the API was not as detailed. We re-used the entire NLP core of PrefFinder without modification. In our initial version, we used the entire preference dump for LIbreOffice, but the runtime was a bit slower than Firefox. We also found that the returned preferences were hard to understand. Since many of the preferences are not available via the menu – our expected use case, we restricted our preferences to only those that are managed by the menu system, which is a subset of the full preference space. To find the proper subset without introducing bias, we first tried all of the menu options in the preferences menu of LibreOffice. We observed which groups of preferences were being modified in the resulting database and then tagged these groups. From this we were able to limit our preferences to a set of 23 categories with 7059 preferences (the list of these preferences can be found on our website). This reduced the search space and helped us to manage our oracle evaluation since that required manual steps as well.

For ABB$_s$ we did not have a way to attach to the runtime database, because this is a production system. Instead we provided a standalone version with a simple GUI. The only customization of the PrefFinder framework was the GUI which took on the order of a few hours to create. Since we had to use the static preferences we obtained from the online documentation system, these had to be translated into a database that PrefFinder could understand. We did this manually and it took about 6 hours to organize and translate. When we started to run queries on the system, we found that there were some common words that the system domain uses as abbreviations which were not in our default general dictionaries. Although our searches were successful, adding these ABB specific abbreviations increased our accuracy, therefore we compiled a list and added this custom set of abbreviations to the database. This iteration added about another hour of development time.

Summary for RQ1. We are able to successfully instantiate and run PrefFinder using the same core for all three systems, albeit with some modifications and programming effort. The primary customization involved programming the connection to the configuration APIs or modeling the configurations from a static location. Other minor customizations were required to refine the quality of the results, such as using only the required subset of preferences for LibreOffice and adding custom database entries for ABB.

5.2 RQ2: Effectiveness of PrefFinder

The effectiveness of PrefFinder is measured by the search success rate (i.e., *accuracy*) and the *execution time.*

Accuracy. The accuracy is shown in Table 4. In this table we see the number of queries run, the number of queries that found the correct answer, and the success rate as a percentage. In the last column we see the range of the ranks where the answer is returned. If the correct answer is returned as the first row of the result, we mark it as a 1. If it is the 10th row it is a 10, etc. When there are multiple preferences specified as an evaluation oracle, we only consider the first one that PrefFinder can find.

For success rate, we see a 76-96% range between systems with ABB having the highest and LibreOffice the lowest success rate. To investigate the ranking distribution of all results (excluding the queries that fail), data is presented in Figure 3 and Table 5. In both the table and the graph we break out our results, showing those in the top 10, those found in position 11-20, etc. As we can see 52% of the queries that return correct results appear in the top 10 rows of the returned results and 72% are returned within the top 30. We believe that using additional lexical information may help to improve these results.

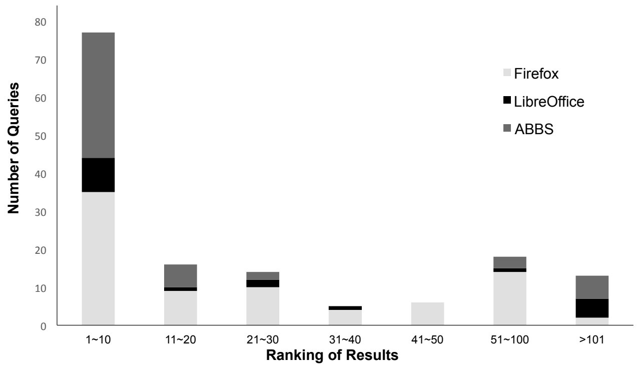


Figure 3: Ranking distribution for Firefox, LibreOffice, and ABB_S

Execution Time. Execution time is the clock time to run PrefFinder once a query is input. We run 100, 25, and 52 user queries for Firefox, LibreOffice, and ABB_S respectively. We run this 10 times to obtain averages. The running time is shown in Table 6. We report the total time to run all questions and the standard deviation (STD). Since we see little variance we only show a single run for the time in our next study. For Firefox it takes on average less than a second to return a result. For LibreOffice it takes about 11 seconds and in ABB_s it takes about one half of a second per query.

We next gathered the time that it took to get answers to individual queries on the user forum and calculated the break even time. Table 7 shows the first 50 queries for Firefox (the rest are on our website) and Table 8 shows all the queries for LibreOffice. When a result was not found, we left the query in the table (shown as fail), and put a dash for break even. A few of the Firefox queries were answered by a *FAQ* which does not have a time stamp so we also use dashes for these cases.

In Firefox, the forum response time ranges from 5 minutes to over 3 months, and its average is nearly 3 days (we only show 50 of them in the table but the results reported are based on all 100 queries, excluding failed ones or the ones that do not have time stamp information). If we suppose it takes about 1 minute (on average) for a user to browse each result returned by PrefFinder until he or she finds the correct preference, then an estimated time that the user spends to get the right answer from typing in a query will be a sum of PrefFinder execution time and the ranking position times 1 minute. For example, if a preference ranked at the 10th place, the lookup time would be about 600 seconds (the execution time is negligible). Based on this assumption, the average lookup time for PrefFinder is only about half an hour, which is less than 1% of the average forum response time. And the forum response time is as high as 817 times large as the PrefFinder lookup time. The break even times for per answer evaluation are from a couple of seconds (in 14% of the questions) to about 2.5 months, and the average time is almost 2 days. More than 33% of the break even time exceeds 10 minutes. We believe that this is artificially high for the real time it would take a user to do this work.

In LibreOffice, the forum response time ranges from 0.3 hours to a couple of days, and the average response time is more than 10 hours. With the same 1 minute browsing time assumption, the average lookup time for PrefFinder is about 3 hours, which is about 33% of the online forum response time. The break even times for per answer evaluation are from a second (only one case) to over 16 hours, and the average time is about 2.5 hours. Almost 60% of the break even time exceeds 10 minutes.

Table 4: Ranking results for 100 Firefox, 25 LibreOffice, and 52 ABB_S user queries

Application	No. Queries	No. Found	Success Rate	Rank Range
Firefox	100	80	80.0%	1 - 140
LibreOffice	25	19	76.0%	1 - 2778
ABB_S	52	50	96.2%	1 - 446

Table 5: Ranking distribution for Firefox, LibreOffice, and ABB_S

Rank Range	1-10	11-20	21-30	31-40	41-50	51-100	>101
Firefox	35	9	10	4	6	14	2
LibreOffice	9	1	2	1	0	1	5
ABB_S	33	6	2	0	0	3	6

Table 6: PrefFinder execution time in seconds for 10 runs on an Ubuntu 12.04.1 machine with an Intel Core i7-2760QM CPU and 5.8G Memory

	No. of Queries	Avg. Tot. Time	STD	Avg. Per Query
Firefox	100	98.5	1.5	0.99
LibreOff.	25	281.0	8.9	11.24
ABB_S	52	27.5	0.9	0.53

Summary for RQ2. PrefFinder is effective at finding user preferences and runs in less than a minute on our queries. When compared with the time it takes to get a new response on a user forum, as long as the user can evaluate what has been returned within 10 minutes per result on average, it is as fast or (in many cases faster) than waiting for a response.

5.3 RQ3: Comparison – Existing Techniques

For our last research question we evaluated Firefox and ABB_S with their existing keyword match mechanisms.

Two sample comparison results are shown in Table 9. For each query we see the individual keywords (in bold) and their results (a fail means that the correct preference was not found). We collate this and show overall results in Table 10. The first row shows the average result ranking for both tools excluding failed queries. As we see, Firefox finds the result

Table 7: Break even timing results for Firefox (first 50 queries)

Query	Rank	Exec (sec)	Forum (min)	Break Even (min)
1	fail	2.29	174416	-
2	6	1.17	37	6.2
3	fail	0.99	9	-
4	fail	0.91	25	-
5	1	1.04	2031	2031.0
6	10	0.89	34	3.4
7	2	0.90	132705	66352.5
8	43	0.88	28	0.7
9	fail	0.83	14	-
10	2	0.93	15	7.5
11	1	0.87	659	659.0
12	fail	0.92	73004	-
13	8	0.93	256	32.0
14	25	1.00	10	0.4
15	56	0.98	36	0.6
16	112	0.93	61	0.5
17	1	0.95	10	10.0
18	26	0.94	117	4.5
19	1	0.91	14569	14569.0
20	22	0.90	55745	2533.9
21	2	0.91	55	27.5
22	62	0.94	189	3.0
23	1	1.08	-	-
24	1	0.92	269	269.0
25	3	0.97	12	4.0
26	fail	0.89	26	-
27	19	0.86	78	4.1
28	77	0.95	221	2.9
29	fail	1.02	85	-
30	fail	1.11	1610	-
31	43	1.14	77	1.8
32	73	1.00	10	0.1
33	26	1.06	245	9.4
34	16	0.93	-	-
35	77	1.07	1914	24.9
36	21	0.96	12	0.5
37	2	1.00	15	7.5
38	fail	0.98	10	-
39	35	0.93	33	0.9
40	28	0.97	114	4.1
41	fail	0.95	203	-
42	2	1.04	9	4.5
43	45	1.02	405	9.0
44	46	0.93	31	0.7
45	51	1.02	34	0.7
46	33	0.93	18	0.5
47	20	0.94	300	15.0
48	15	0.98	29	1.9
49	4	1.02	61	15.3
50	30	0.94	12	0.4

Table 8: Break even timing results for LibreOffice

Query	Rank	Exec Time (sec)	Forum (min)	Break Even (min)
1	1	11.1	312	312.0
2	fail	10.9	689,626	-
3	10	10.0	191	19.1
4	336	11.1	61	0.2
5	130	10.9	168	1.3
6	fail	10.4	8	-
7	fail	10.4	300	-
8	2	10.7	1976	988.0
9	1	10.4	58	58.0
10	14	10.5	94	6.7
11	116	11.2	245	2.1
12	1	10.7	211	211.0
13	29	11.8	474	16.3
14	2	11.6	56	28.0
15	2778	11.5	103	0.0
16	75	11.0	6007	80.1
17	40	10.5	16	0.4
18	1	12.2	975	975.0
19	fail	11.3	21	-
20	fail	11.2	10	-
21	fail	10.4	171	-
22	5	10.8	51	10.2
23	21	10.8	175	8.3
24	1	11.7	91	91.0
25	296	12.3	458	1.5

(on average) at the 27.1th postion, while *about:config* finds it a the 12.9th position. The second row shows the success rate for either the full prefFinder query or when *at least one* keyword is found for that query. For example, we would say that both of the example queries in Table 10 pass since at least one keyword find the results. PrefFinder returns the correct answer 80% of the time, while `about:config` does so only 64% of the time.

We aslo report in parentheses the number of times an individual keyword finds the correct answer. Using the example queries in Table 10 this would be 2 out of 6 or 33% of the time. We see that using `about:config` with each individual extracted keyword returns the correct answer only 30% of the time. This tells us that more than two thirds of the time, a keyword failed to find the answer confirming what we know (that it is sensitive to keyword selection). The last row of this table shows the number of successful searches for each tool, where the other tool failed. There are 20 queries where PrefFinder succeeded and `about:config` failed, while only 4 where `about:config` found an answer and PrefFinder did not.

We next compare to the keyword match utility in ABB_{D1}. The comparison results are shown in Table 11. We first show results for the exact query *Query*, next we show a combination of *all* keywords extracted from the query, and then each *individual* keyword. The first row shows the average ranks, with failed queries excluded. When all keywords are used together we get the best average rank (1.8). The next best rank is for the Individual keyword searches, followed by PrefFinder (12.4) and finally by the full query (23.0). If we next look at success rate we see that when we use individual keywords, we find the correct answer for at least one keyword in a query 98.1% of the time. The individual rate of success for keywords is lower (84.7%), and the full set of keywords only has a 74.1% success rate both of which are lower than PrefFinder (96.2%). The worst scenario is when we type in the full query for only a 14.8% success rate. Finally, we show two rows that tells us (1) how many queries pass in PrefFinder, but fail in the other techniques (i.e. 21 of the full Query, 2 of the All keywords and none of the Individual ones) as well as the reverse (how many queries succeed in the other technique but fail in PrefFinder – a total of 4 across all techniques). We conclude from this, that the keyword match utilities find the answer at a lower rank when

Table 9: Sample results: PrefFinder vs. Firefox about:config (up to 3 keywords)

Query	PrefFinder Rank	about:config Rank		
(1) When closing Firefox windows, I would like a warning before the last window closes.	8	**windows** Fail	**warning** 3	**closes** Fail
(2) How do I prevent the warning for closing multiple tabs at once from displaying?	1	**warning** Fail	**closing** Fail	**tabs** 19

successful, but overall they are more sensitive to failure. We believe that the high success rate of ABB$_{D1}$ has to do with the way that the queries were written (and keywords hence extracted). The writers of these queries are expect engineers and they will tend to use forms of the words and terminology consistent with the preference system. In addition, the keyword match utility in ABB can also search the full document (instead of just preferences). This might lead to better success, but also has the potential for more noise.

Table 10: PrefFinder vs. `about:config`

	PrefFinder	`about:config`
Avg. Rank	27.1	12.9
Success Rate (%)	80	64 (30)
Succeeded: Other failed	20	4

Table 11: PrefFinder vs. ABB_{D1} `keyword match`

	PF	`keyword match` Query	All	Indiv.
Avg. Rank	12.4	23.0	1.8	2.6
Success Rate (%)	96.2	14.8	74.1	98.1 (87.4)
Pref Succeed: Other failed	-	21	7	0
Other Succeed: Pref failed	-	0	2	2

Summary for RQ3. PrefFinder is competitive against existing keyword search utilities. The success rate is as good or higher than these systems, and allows for more noise, although the accuracy when successful is slightly lower.

6. RELATED WORK

There has been a lot of research related to sampling and reducing the large configuration space for testing and maintenance of software [7, 32, 42], for prioritizing these samples [28, 36] and for change impact analysis [27]. At the code level, symbolic execution has been used to identify dependencies and viable interactions of preferences for testing [31, 35], or to perform analysis that is configuration-aware [15,17]. From a reverse engineering perspective, Rabkin and Katz extract source code level preference names [30]. Other work aims to fix (or diagnose) problems when configurations fail [3, 38, 40, 41, 43, 44]. This thread of work requires that some unwanted behavior has been observed (a misconfiguration) and can be recreated. The aim is not to search for individual configuration options, but to return the system to a non-faulty state. ConfSuggester by Zhang and Ernst [45] is the most similar work to ours in that is searches for a single configuration option (from the source code) to return to the user. However ConfSuggester still requires the user to demonstrate the different behavior (it only works for regressions where some default behavior has changed) and is limited to Java. Furthermore it requires instrumentation at the byte code level which means it is language dependent. PrefFinder can be used to search for configurations without demonstration of altered behavior and does not depend on the underlying programming language .

There has been considerable work on using natural language to improve code documentation and understanding [8,9,11,12,34] and to create code traceability links [5,18,26]. In addition, recent work on finding relevant code, uses search to find code snippets that satisfy a given purpose [20, 37]. While this work is related to our problem, the techniques assume that there is a large code base to explore and leverage this in their similarity techniques; we want to associate behavior with identifier names with little or no context.

PrefFinder is unique in that searches highly configurable multi-lingual software systems without access to the code, and uses only natural language. The return result is the preference name that is used within the main preference database (a higher level of abstraction than a variable name). It can also connect with an application's preference system and documentation at runtime.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented PrefFinder, a natural language framework for finding preferences in highly configurable software. PrefFinder uses a core NLP engine, but is customizable at both the application and database end. We instantiated three versions of this framework and performed a case study on two large open source and one industrial application. We find that prefFinder is able to find the right preferences for as many as 96% of the queries, but no less than 76% using only seconds of time. When we compared the time taken to find the same answers when newly asked on a user forum and calculated a break even point, we found that it would require a user (on average) more than 10 minutes per answer evaluation time to make PrefFinder the slower alternative. And when we estimate a time of one minute per answer we see as much as an 800% improvement in time. When compared with existing keyword match utilities we find that PrefFinder is more robust, albeit the rank of the returned result may be lower.

In future work we will extend PrefFinder to handle additional use cases. First we will consider constraints between options and multiple options for a single query. We will continue to work on the accuracy of our results by adding additional lexical information and will develop traceability links between the menu, preference names and code. We will also include additional artifacts such as code and documentation to enrich the knowledge returned back to the user. Finally, we plan to evaluate PrefFinder in user studies.

Acknowledgments

This research was supported in part by the National Science Foundation awards CCF-1161767 and CNS-1205472, and the Air Force Office of Scientific Research award FA9550-10-1-0406.

8. REFERENCES

[1] Computer acronyms list. `http://www.francesfarmersrevenge.com/stuff/archive/oldnews2/computeracronyms.htm`.

[2] AskLibO. `http://ask.libreoffice.org/`, 2014.

[3] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 1–11, 2010.

[4] D. Binkley and D. Lawrie. Development: Information retrieval applications. In *Encyclopedia of Software Engineering*, pages 231–242. 2010.

[5] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol. Can better identifier splitting techniques help feature location? In *International Conference on Program Comprehension (ICPC)*, pages 11–20, 2011.

[6] LibreOffice. `http://libreoffice.org/`, 2013.

[7] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 243–253, 2011.

[8] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *International Working Conference on Mining Software Repositories, MSR*, pages 71–80, 2009.

[9] H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *In Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006)*, 2006.

[10] Ispell. `http://www.gnu.org/software/ispell/`.

[11] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *International Working Conference on Mining Software Repositories (MSR)*, pages 79–88, 2008.

[12] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically mining software-based semantically-similar words from comment-code mappings. In *Working Conference on Mining Software Repositories*, May 2013.

[13] D. Jin. Improving Preference Recommendation and Customization in Real World Highly Configurable Software Systems. Master's thesis, University of Nebraska-Lincoln, Lincoln, NE, USA, August 2014.

[14] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *International Conference on Software Engineering Companion Volume, Software Engineering in Practice*, SEIP, pages 215–224, 2014.

[15] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D'Amorim. SPLat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 257–267, New York, NY, USA, 2013. ACM.

[16] S. E. Kristian Wiklund, Daniel Sundmark and K. Lundqvist. Impediments for automated testing - an empirical analysis of a user support discussion board. In *International Conference on Software Testing, Verification and Validation*. IEEE, 2014.

[17] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 81–91, New York, NY, 8 2013.

[18] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), Sept. 2007.

[19] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[20] C. McMillan, M. Grechanik, D., C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, Sept. 2012.

[21] G. A. Miller. WordNet: A lexical database for english. *Communications of the ACM*, 38:39–41, 1995.

[22] mozdev.org. `http://preferential.mozdev.org/preferences.html`, 2014.

[23] XPCOM. `https://developer.mozilla.org/en-US/docs/XPCOM`.

[24] Mozilla Support. `https://support.mozilla.org/`, 2013.

[25] mozillaZine Knowledge Base. `http://kb.mozillazine.org/Knowledge_Base`, 2014.

[26] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. D. Lucia. When and how using structural information to improve IR-Based traceability recovery. In *European Conference on Software Maintenance and Reengineering, CSMR*, pages 199–208, 2013.

[27] X. Qu, M. Acharya, and B. Robinson. Impact analysis of configuration changes for test case selection. In *International Symposium on Software Reliability Engineering*, ISSRE, pages 140–149, 2011.

[28] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *International Symposium On Software Testing and Analysis*, pages 75–86, 2008.

[29] A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *International Conference on Automated Software Engineering (ASE)*, pages 193–202, nov 2011.

[30] A. Rabkin and R. Katz. Static extraction of program configuration options. In *International Conference on Software Engineering*, ICSE, pages 131–140, 2011.

[31] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *International Conference on Software Engineering*, ICSE, pages 445–454, 2010.

[32] B. Robinson and L. White. Testing of user-configurable software systems using firewalls. *International Symposium on Software Reliability Engineering, ISSRE*, pages 177–186, Nov. 2008.

[33] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. In *Information Processing and Management*, pages 513–523, 1988.

[34] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *International Conference on Aspect-oriented Software Development*, pages 212–224, 2007.

[35] C. Song, A. Porter, and J. S. Foster. iTree: efficiently discovering high-coverage configurations using interaction trees. In *The International Conference on Software Engineering*, ICSE, pages 903–913, 2012.

[36] H. Srikanth, M. B. Cohen, and X. QU. Reducing field failures in system configurable software: Cost-based prioritization. In *International Symposium on Software Reliability Engineering, ISSRE*, pages 61–70, Nov 2009.

[37] K. T. Stolee and S. Elbaum. Toward semantic search via SMT solver. In *International Symposium on the Foundations of Software Engineering (FSE)*, pages 25:1–25:4, 2012.

[38] Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving configuration management with operating system causality analysis. *SIGOPS Operating Systems Review*, 41(6):237–250, Oct. 2007.

[39] Sun Microsystems. *OpenOffice.org 3.1 Developer's Guide*. Sun Microsystems, 2009.

[40] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI, pages 77–90, 2004.

[41] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *International Conference on Software Engineering*, ICSE 2012, pages 58–68, 2012.

[42] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, Jan 2006.

[43] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Symposium on Operating Systems Principles*, SOSP, pages 159–172, 2011.

[44] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *International Conference on Software Engineering*, ICSE, pages 312–321, 2013.

[45] S. Zhang and M. D. Ernst. Which configuration option should I change? In *International Conference on Software Engineering*, ICSE, pages 152–163, 2014.