Extracting Meaning from Abbreviated Identifiers

Dawn Lawrie Henry Feild David Binkley
Loyola College
Baltimore MD
21210-2699, USA
{lawrie, hfeild, binkley}@cs.loyola.edu

Abstract

Informative identifiers are made up of full (natural language) words and (meaningful) abbreviations. Readers of programs typically have little trouble understanding the purpose of identifiers composed of full words. In addition, those familiar with the code can (most often) determine the meaning of abbreviations used in identifiers. However, when faced with unfamiliar code, abbreviations often carry little useful information. Furthermore, tools that focus on the natural language used in the code have a hard time in the presence of abbreviations. One approach to providing meaning to programmers and tools is to translate (expand) abbreviations into full words. This paper presents a methodology for expanding identifiers and evaluates the process on a code based of just over 35 million lines of code. For example, using phrase extraction, fs_exists is expanded to file_status_exists illustrating how the expansion process can facilitate comprehension. On average, 16 percent of the identifiers in a program are expanded. Finally, as an example application, the approach is used to improve the syntactic identification of violations to Deißenböck and Pizka's rules for concise and consistent identifier construction.

Keywords: Software Quality, Program Identifiers, Program Comprehension

1 Introduction

The first *Working Session on Information Retrieval Based Approaches in Software Evolution* was held at ICSM'06. The number of attendees at this session underscores the growing interest in applying Information Retrieval (IR) techniques to software engineering problems. In order to make effective use of IR techniques, consistent meaningful vocabulary is important. Identifiers present a challenge when domain information is buried behind abbreviations and acronyms. One way to expose this information is to translated terse identifiers into natural language. Then existing (and future) IR based techniques can be employed by software engineering tools to make use of this information in problems such as concept location [22], quality assessment [16], and software reuse [8].

The automated translation of existing identifiers to natural language is complicated by many facets. For example, identifiers often include prefixes, suffixes, or use 'common' abbreviations (believed to be so well known that they go undocumented). To address these, the translation process applies three IR based techniques. First, probability distributions such as relative entropy [19] are used to identify *usual* versus *unusual* aspects of identifiers.

Second, for specific parts of each identifier (*e.g.*, non-dictionary words), sets of possible expansions are generated based on wildcard expansions (expansions that begin with the same letter as the abbreviation and subsequent letters occur in the same order). A learning algorithm is applied to determine the likelihood of a particular expansion based on characteristics such as the proportion of adjacent letters in the expansion. For example, lib can expand to library and liberty. In a function that includes comments such as "find opponent liberty and choose move to attack", the expansion would favor liberty over library.

Finally, for multi-word identifiers, machine translation techniques are employed to ensure that the expansion has produced a meaningful cohesive unit. For example, the identifier thenewestone includes three dictionary words fused together. This identifier is a challenge for a naive splitting algorithm as it can be split into three dictionary words two different ways: the-newest-one or then-ewe-stone. Borrowing from machine translation techniques for resolving translation ambiguity, the likelihood of finding the words the, newest, and one versus that of finding then, ewe, and stone in close proximity can be compared.

The usefulness of expanded identifiers is far reaching. For example, they could be used to provide GUI 'tool tips', as an aide to an engineer who must map the program's

identifiers to the concepts that they represent. Furthermore, by increasing the amount of natural language found in the code, expansion improves existing source code analysis tools that attempt to map program identifiers to domain level concepts (*e.g.*, those appearing in the documentation). Examples of this include the techniques of Antoniol et al. and Marcus et al. that seek to (re)establish links between source code and its documentation [3, 21]. In another example expansion allows for better automatic recognition of quality identifiers: while humans can often comprehend code whose identifiers are composed of meaningful abbreviations [17], tools have a much more difficult time doing so. Finally, expansion facilitates correctly identifying violations of conciseness and consistency rules for identifiers such as those laid out by Deißenböck and Pizka [7, 15]. This final example is used herein to demonstrate an application of the expansion.

This paper presents a first step in this process: the expansion of abbreviations and acronyms into meaningful words and phrases based on local function-level language information. For example, a phrase dictionary, built for each function using phrases found in the comments, provides one source of meaningful names. This step allows the value of local information to be better understood and also highlights places where using global information is useful.

The remainder of this paper consists of a description of the process used to expand identifiers in Section 2. An analysis of the expanded identifiers is presented in Section 3. Section 4 describes the example application of the technique. The final sections discuss related and future work and conclude.

2 Technique

In the current implementation, identifier expansion is a two step process. The first step, described in Section 2.1, splits an identifier into its constituent *words*. Then, as explained in Section 2.2, a collection of possible expansions are considered.

2.1 Identifier Splitting

Following past studies, identifiers are first divided into their constituent parts for analysis [3, 7, 6, 9, 25]. Herein, these parts are called "*words*" – sequences of characters with which some meaning may be associated. Two kinds of words are considered: *hard words* and *soft words*. Hard words are separated by the use of word markers (*e.g.*, the use of camelCasing or under_scores) [3]. For example, the hard word spongeBob and sponge_bob both contain the well separated hard words sponge and bob.

For many identifiers, the division into hard words is sufficient. This occurs when all hard words are dictionary words or known abbreviations. When a hard word is in neither category, the identifier may contain non-well-separated words. For example, the identifier easycase includes the two non-well separated words easy and case. The identification of these "*soft words*" is the goal of identifier splitting. One approach does so using a greedy algorithm that recursively finds the longest prefix and suffix that are in the dictionary or a known abbreviation list [9].

For example, the identifier zeroindeg includes a single hard word because there are no word markers; thus, division into hard words is insufficient to identify the concepts within the identifier. The splitting algorithm divides this hard word into the three soft words zero-in-deg.

2.2 Expanding Abbreviations

The expansion algorithm works independently on soft words in the context of the source code for a particular function. It uses four lists of potential expansions: a list of natural-language words extracted from the code, a list of phrases extracted from the code, a list of programming language specific words referred to as a *stoplist*, and finally a natural-language dictionary. There are two sources of words for the first list: first, words contained in the comments that appear before and within the function, and second dictionary hard words found in the identifiers of the function.

The phrase list is obtained by running the comments and multiword-identifiers through a phrase finder [10]. Here, the first letter of each word in the phrase is used to build an acronym. If a sequence of letters matches an acronym exactly, the phrase is considered a potential expansion. For instance, the phrase finder extracts the phrase *file status* from the comments of the program which. This phrase is later returned as the expansion for the soft word fs contained in the identifier fs_exists.

Once the list of potential words and phrases has been extracted, expansion begins. This process has two stages that look first in the word list extracted from the code and comments and in the phrase dictionary, and then in a natural-language dictionary. Thus, language extracted from the source is favored over words that appear in the dictionary, and words on the stoplist are treated the same as natural language words. A word from one of these list matches an abbreviation if it begins with the same letter and the individual letters of the abbreviation occur, in order, in the word. For example, the search for "abs" in the mozilla source correctly discovers the expansion absolute, while horiz and triag are correctly expanded to horizontal and triangle, respectively.

In the present implementation, if there is a single match in either stage of the search, it is returned as the expansion of the abbreviation. Future work will explore techniques for selecting from among multiple potential expansions by fa-

voring words where the sequences of letters are adjacent, minimizing the number of different expansions for a sequence within the program and across programs, and, for multi-word identifiers, maximizing the likelihood that the words occur together based on co-occurrence information derived from approximately 1 trillion word tokens of text from publicly accessible web pages. This data set of n-grams was developed by Google and is available from the Linguistic Data Consortium [4].

Two information retrieval (IR) techniques are used to improve the extraction. The first technique, *stemming*, eliminates word suffixes; thus, ignoring the particular form of a word [13]. For example, the 'stem' of '*walk*', '*walking*', and '*walked*' is '*walk*'. Stemming is used to more accurately determine if hard words are dictionary words, since the particular type of stemming used (Krovetz stemming) stems words to dictionary words [13].

The second technique, *stopping*, filters soft words through a *stop-list*, which in IR is a collection of words not thought to be relevant to any query. For example, in English, words such as '*the*' and '*an*' are stop-list words. When considering source code, a special stop-list is used that includes programming language specific entries. For example, the stop-list for C includes keywords (*e.g.*, `while`), predefined identifiers (*e.g.*, `NULL`), library function, and variable names (*e.g.*, `strcpy` and `errno`). Because the meaning of these stop words are rarely described in the code and engineers generally recognize them as is, it would be inadvisable for the expansion algorithm to modify them, for example, by expanding `strcpy` to `string_copy`.

3 Analysis

This section presents an analysis of the expansion algorithm. It begins by discussing the 158 programs studied, and then, to illustrated the successes and failures of the algorithm, considers two representative examples (the programs which and a2ps) in greater detail. Finally, the quality of the expansion is considered first by comparing the tool generated expansion of 64 identifiers with 'by-hand' expansion of the same identifiers and then using quantitative statistics.

3.1 Subject Programs

The analysis includes empirical data collected from 158 programs, some of which are different versions of the same program. Ignoring multiple versions, 63 unique programs were considered. Programs ranged in size from 1,423 to 3,003,526 LoC and covered a range of application domains (*e.g.*, aerospace, accounting, operating systems, program environments, movie editing, games, etc.) and styles (command line, GUI, real-time, embedded, etc.). Most of the code was written in C. Significant C++ and Java code were also studied.

Table 1 shows 10 representative programs. It reports code sizes for the C, C++, and Java code (and their sum) as counted by the Unix utility wc (excluding header files). In addition, the total number of non-comment non-blank lines of code, as reported by sloc [27], is shown. The average percentage of non-comment non-blank lines varies by language with 66% of the C code, 72% of the C++ code, and 58% of the Java code being non-comment non-blank lines.

Table 2 summarizes statistics regarding the identifiers along with some demographic information (*e.g.*, dominant programming language, and the start and release years of the program). The table shows a representative sample of the programs. Finally, Table 3 summarizes identifier related data over all programs (not just those shown in Tables 1 and 2). The table presents that data by programming language and all the data taken collectively.

3.2 Example Output

Using several examples, this section illustrates the output of the expansion process. The examples, taken from the programs which and a2ps, highlight the capabilities and weaknesses of the current algorithm. These include correct expansions, incorrect expansions of valid abbreviations, and finally incorrect expansions due to hard word splitting errors.

To begin with, the top three entries in Table 4 show some triumphs of the algorithm. Each example uses a different source to find the correct expansion. In the case of the first identifier, pw_dir, the word *password* is found in a comment near the appearance of the identifier. As an aside, dir is not expanded because there are three possible expansions, one of which is *directory*. With the second example, the phrase finder discovers the correct expansion. The code includes the identifier file_status. The phrase finder enters *file status* as a phrase having the acronym fs. Therefore, the expansion, when trying to match FS uncovers the phrase *file status*. In the third example, the word *direction* is used in an identifier within the function that includes the abbreviation STACK_DIR. Thus, here again the code itself is the source of the expansion and STACK_DIR is correctly expanded to STACK_DIRECTION.

This last example is interesting as it illustrates the need for context in the expansion process. The example comes from the file alloca.c. In this file dir corresponds to *direction* and is correctly expanded to direction. However, in the file tilde.c of the same program, the abbreviation dir correctly expands to directory. From a comprehension standpoint, this is an issue if these identifiers are visible in the same scope (for example, if one were a global). The issue is less severe if both are local to their own module (file or

program	wc C	wc C++	wc Java	wc Total	sloc Total
cinelerra-2.0	1,044,996	106,357	0	1,151,353	820,980
cpm68k1-v1.2a	132,171	0	0	132,171	102,252
empire_server	85,548	0	0	85,548	62,793
eMule0.46c	1,759	172,164	0	173,923	135,567
ghostscript-7.07	302,336	2,872	0	305,208	225,459
jakarta-tomcat-5.5	68,003	0	353,604	421,607	219,766
LEDA-3.0	41,610	0	0	41,610	27,425
minux-2.0	326,210	0	0	326,210	244,033
mozilla-1.4	1,047,741	1,949,292	6,493	3,003,526	2,107,436
quake3-1.32b	353,806	57,431	0	411,237	281,432
Totals for all code not just that shown above					
total	26,338,235	15,374,576	6,909,487	48,643,480	32,521,078

Table 1. Subject Programs.

program	dominant language	start year	release year	LoC (wc)	unique ids	id instances	hard words	soft words	percent‡ increase
apache_1.3.29	C	1995	1999	101,515	8,419	99,172	17,942	22,971	28.0%
cinelerra-2.0	C	1996	2004	1,151,353	84,612	1,833,424	209,059	261,793	25.2%
cpm68k1-v1.1	C	1974	1983	73,172	4,167	79,660	4,560	8,193	79.7%
cvs-1.11.1p1	C	1989	2001	91,446	5,876	82,315	10,918	14,344	31.4%
eclipse-3.2m4	Java	2001	2005	3,087,545	167,662	3,893,272	554,068	612,632	10.6%
gcc-2.95	C	1987	1999	841,633	44,941	897,728	110,060	146,474	33.1%
jakarta-tomcat-5.5.11	Java	1999	2005	421,607	19,202	351,487	48,537	54,471	12.2%
minux-2.0	C	1980	1996	326,210	22,951	325,341	35,967	51,428	43.0%
mozilla-1.6	C++	1998	2004	2,919,307	189,916	3,649,329	563,448	659,396	17.0%
mysql-5.0.17	C++	1996	2005	1,293,270	50,383	1,023,362	132,249	163,363	23.5%
quake3-1.32b	C	1999	2005	411,237	31,114	542,664	75,474	94,144	24.7%
sendmail-8.7.5	C	1983	1996	78,757	2,877	62,075	4,492	6,828	52.0%
spice3f4	C	1985	1993	298,734	12,388	452,423	24,599	34,882	41.8%

Table 2. Basic counts from 14 selected programs. Some of the programs from Table 1 are repeated for comparison, other's were selected to provide diversity in the presented data. ‡Percent increase is the percent increase from hard words to soft words.

function); although even in this case a programmer considering both modules may still confuse the two.

The second grouping of three identifiers in Table 4 shows examples where the algorithm chose the wrong expansion. Most such problems occur when an abbreviation is so common that it is unlikely the full word expansion appears in the code or comments. For example, in the first case cnt corresponds to *count*. In the second example, da is misinterpreted as dictionary due to a lack of good information in the local context. Looking in the file where the function da_qsort is defined it is obvious that da should match the phrase *dynamic array* (something the phrase finder gets correct within the file). However, the error occurs in the file where da_qsort is called. This example is indicative of a category of false positives that would be mitigated by considering a broader scope (*e.g.*, the entire program). Unlike the expansion of dir above, because da_qsort is in the global scope, having a single expansion is preferable. The final example is particularly interesting case because r and n refer to the whitespace characters '\r' and '\n' rather than natural language words. In this case r and n are rather informative and thus expansion has little to offer.

The last grouping of three identifiers in Table 4 demonstrates problems in the splitting that lead to problems in the expansion. In the first case, argcasematch should have been split arg_case_match. Unfortunately, argc is an abbreviation known to the splitter while arg is not, and therefore, the split that includes argc is preferred. In the second case, the splitting algorithm did not recognize any of the subsequences so the splitting is, in essence, random, and the words chosen for expansion are nonsensical. Had the splitter produced xv_str_rpl instead, then using comments and

Totals for (over all code not just that shown)	instances per id	hard words per id	soft words per id	LoC (wc)	unique ids	id instances	hard words	soft words	percent increase
C	18.6	2.5	3.1	26,338,235	1,566,289	2,9074,119	3,956,372	4,821,045	21.9%
C++	19.3	2.9	3.5	15,375,576	965,402	18,836,801	2,835,896	3,341,987	17.8%
Java	22.1	3.0	3.4	6,909,487	356,225	7,885,428	1,076,709	1,203,537	11.8%
Grant Total	19.3	2.7	3.2	48,643,480	2,890,153	55,638,621	7,872,119	9.370,562	19.0%

Table 3. Total counts from all programs.

	Original Identifier	**Split Identifier**	**Expanded Identifier**
1	pw_dir	pw_dir	password_dir
2	FS_EXISTS	fs_exists	file_status_exists
3	STACK_DIR	stack_dir	stack_direction
1	cnt	cnt	current
2	da_qsort	da_qsort	dictionary_qsort
3	eol_rn	eol_rn	eol_Returns
1	argcasematch	argc_a_se_match	argc_a_size_match
2	xvstrrpl	xv_st_rr_pl	xv_stream_rr_Plain
3	EDF_SUFFIX	ed_f_suffix	encodings_f_suffix

Table 4. Example Identifiers

source from the file where the function is defined leads to the correct expansion xv_string_replacement. In the final example, no splitting should have occurred. The acronym *EDF* is defined in the preceding comment as *Encoding Description Files*. One of the words shows up in the expansion, but this was luck. This last group of examples points to the need for better integration between the splitting and expansion algorithms. In all three cases such integration would have improved the splitting and the subsequent expansion.

3.3 Manual Inspection

To gain a feeling for the quality of the automatic expansion, a manual inspection of the output was performed. Considering millions of identifiers was not practical, so a random sample of 64 identifiers effected by expansion was created. For each, the correctness was judged by three people: one of the three authors (randomly chosen) and two student programmers at the end of their second or third year of study. The inspection involved being given an identifier before and after expansion along with the function (including preceding and internal comments). If the expansion was deemed incorrect, then the programmer was asked to provide a correct expansion.

The collected data was filtered so that each identifier had a single judgement. Where discrepancies occurred, the majority judgment was used. In most cases there were three judgements for each identifier. However, because two students misunderstood the directions, their data was disregarded. This resulted in five ties, in which cases the authors decided the correct response; thus, 64 identifiers were considered in the remainder of the inspection. About one-third of the identifiers were correctly split and expanded. Surprisingly, 36 of the expansions were based on a single letter. One difficulty with expanding single letters is that they are not always an abbreviation, such as when n represents the columns in a 2-dimensional array.

The first inspection aims to ascertain the correctness of the expansion algorithm independently of the splitting algorithm. Thus, only identifiers that are split correctly are considered here. Overall the data, 58% of the identifiers are expanded correctly. For identifiers whose expansions were based on one or two characters, 57% of the identifiers are expanded correctly; however, for identifiers whose expansion were based on three or more characters, the percentage rises to 64%. Thus, it appears that longer soft words lead to increased accuracy.

The second inspection considers the correctness of the splitting algorithm. A similar pattern is observed. Overall the data, 75% of the identifiers are correctly split. For identifiers of one or two characters grouped together, 57% are split correctly. The percentage rises to 91% for identifiers with groups of three or more characters. This indicates that when splitting algorithm groups a large number of characters, it is more likely to have split the identifier correctly.

In addition to the quantitative analyses, qualitative analysis of the data suggests several ways the algorithm can be improved. First, many of the incorrect expansions are caused by prefixes (*e.g.*, m_size for the attribute (member) size). In such cases, the expansion algorithm should expand these prefixes uniformly throughout the entire program and be conservative by only expanding when there is strong evidence in favor of the expansion. Second, standard acronyms such as *RDF* and *CERT* tend not to be defined in the code. Such acronyms (as opposed to abbreviations used in a particular program) are more likely to appear in external documentation. This makes their recognition more of a challenge; however, capitalization may be helpful.

3.4 Quantitative Statistics

This section evaluates the impact of the expansion algorithm on the entire code base. Four different statistics are considered. The first two focus on the identifiers that are expanded using the current algorithm. The second two ex-

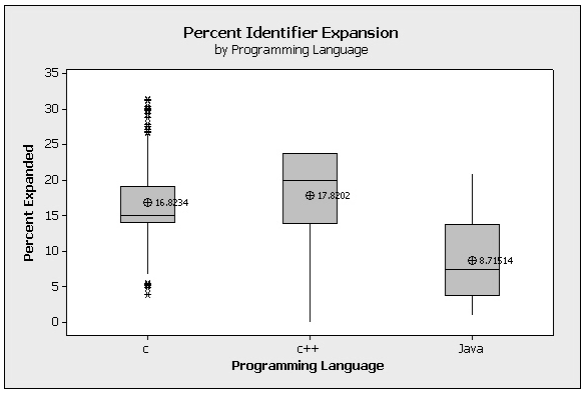

Figure 1. Percentage of expanded identifiers per program by programming language.

amine all the identifiers and the possibility of expansion and the usefulness of different sources for expansion.

Overall, about 16% of identifiers in programs are modified by the current expansion algorithm. By program, this value ranges from 3% to 31%. Figure 1 shows a boxplot of the data by programming language. Given that a majority of the subject programs are written in C, it is no surprise that its characteristics are similar to the overall characteristics. It is interesting to note that on average, more expansions occur for C++ code, while fewer occur for Java code. Given that both are object oriented languages, this is probably a result of the strict naming conventions that Java programmers are encouraged to use.

Given that the expansion algorithm uses local information for expansion, an identifier, such as a global variable or function name, that appears in multiple functions is not necessarily expanded the same in all contexts. For example, pd is expanded to printed and published. An analysis of the number of identifiers involved in multiple expansions was performed to determine the extent of this effect. On average 7% of identifiers have multiple expansions (*i.e.*, just under half the expanded identifiers have multiple expansions). This indicates the importance of using more global information to provide a more consistent expansion of identifiers. By program, this percent ranges from less than one percent to 16%.

Thus far, the focus of the evaluation has been on the identifiers that contain at least one soft word with a single possibility for expansion. The second part of the discussion includes all identifiers. By considering all the soft words, one can understand the scope of the expansion task. In total 57% of the soft words are associated with a single word. Included in this percentage are all the words recognized as dictionary words or stoplist words. These account for 50% of the soft words (20% are stoplist words and the remainder come from the dictionary). This means that only 7% of the soft words are expanded.

In order to ascertain the number of possibilities that the machine translation portion of the algorithm will have to cope with, soft words can be ranked by size of the sets of possible expansions. Of the 8.1 million unique soft words studied, sets range in size from 1 to 6735 possible expansions. Although the largest sets are much larger than traditional machine translation algorithms consider, 25% of the soft words requiring expansion have 10 or fewer possibilities, which is a reasonable size for these sorts of algorithms. After size 10, sets grow in size rapidly. For example, considering 30% of the soft words includes sets of size 140.

When considering the non-dictionary soft words, the sources for the expansions help explain the usefulness of each source. Table 5 shows the break down of the sources for soft words that are expanded by the current algorithm and soft words that are not. Note that the percentages on each line do not necessarily add to 100% because a single soft word might have multiple sources for expansion. In addition, even when a soft word is in the dictionary (making it a *Word* in the Table), the comments and source code are still examined for expansions as short words are sometimes used as abbreviations such as *cat* which abbreviates *concatenation* rather than the referring to the animal. Finally, none of the abbreviations with a single expansion are words because such soft words would not be expanded.

Encouragingly, the source code and documentation provide a possible expansion for over 60% of soft words. In addition the dictionary is not currently being utilized very frequently for soft words with a single expansion. When examining multiple word sets, dictionary usage goes up to 36%. The code and comments share equally in providing possible expansions which is another indication of the extend of natural language embedded in identifiers. Finally, phrases are not used very frequently which may indicate that acronyms are less likely to be defined in the code.

From the analysis of potential expansions, considering soft words with ten of fewer expansions allows 25% of the soft words to be expanded. Therefore, the last two lines of Table 5 separate soft words into those with *small multi-word sets*–sets of sizes between two and ten, and those with *large multi-word sets*–sets of size greater than ten. The percentages for small sets are much more similar to the singleton sets. In particular, the dictionary is only used 2% of the time. Furthermore, for small sets, the code and documentation are good sources for expansions of soft words. In the large sets, the dictionary is required 87% of the time, indicating the dictionary is generally responsible for the large number of possible expansions.

	Code	Comment	Phrase	Word	Dictionary
Singleton Sets	43%	51%	2%	0%	3%
Multi-Word Sets	52%	46%	2%	30%	36%
Small Multi-Word Sets	77%	67%	2%	50%	2%
Large Multi-Word Sets	13%	13%	0.3%	0.4%	87%

Table 5. Sources of expansions for soft words

4 An Application

Multiple studies have pointed to the importance of good identifier names. For example, Rilling and Kelmola observe "In computer programs, identifiers represent defined concepts" [25], while Caprile and Tonella point out that "Identifier names are one of the most important sources of information about program entities" [6].

Concise and consistent variable naming can improve identifier naming and thus code quality; thus, producing programs that are easier to comprehend and manipulate due to a lack of naming confusion. To illustrate the usefulness of abbreviation expansion, its impact on the detection of identifier conciseness and consistency failures is considered [7, 15].

The original detection algorithm requires a mapping from the domain of identifiers to the domain of concepts [7]. A *syntactic* version of this algorithm, which replaces the need for a concept mapping with information extracted from the syntactic makeup of the identifiers, fails to identify violations what occur in the presence of abbreviations within identifiers [15].

The expansion algorithm presented herein can be used to improve the syntactic approach. The presentation of this idea begins in Section 4.1, which introduces Deißenböck and Pizka's rules for concise and consistent identifier construction [7] and a syntactic version of these rules [15]. It then, in Section 4.2, considers the impact of abbreviation expansion on the syntactic rules is considered.

4.1 Concise and Consistent Identifiers

Deißenböck and Pizka describe a formal model for well-formed identifier naming that includes rules for *consistent* and *concise* naming of identifiers [7]. Their rules are based on a mapping of identifiers to the set of all concepts used in a program. Thus, they provide "a formal model based on bi-jective mappings between concepts and names"[7].

Their rules include three requirements: two related to identifier consistency (involving homonyms and synonyms) and one related to identifier conciseness. These three are formalized as follows: an identifier *i* is a homonym if it represents more than one concept from the program (*e.g.*, the identifier file in Figure 2a). Two identifiers *i1* and *i2* are synonyms if the concepts associated with *i1* have a non-empty overlap with the concepts associated with *i2* (*e.g.*, the identifiers file and file_name share the concept *file name* in Figure 2b). The presence of homonyms or synonyms indicate inconsistent naming of concepts in a program and thus violate Deißenböck and Pizka's identifier consistency rule. Finally, an identifier *i* for concept c is concise if no concept less general than c is represented by another identifier. For example, the identifier position most directly corresponds to the concept *position*. It would concisely represent the concept *absolute_position* provided that the program did not include any other position concepts that were less general then *position* (*e.g.*, *relative position*).

In the absence of a mapping from identifiers to concepts, a restricted form of synonym consistency and conciseness, referred to herein as *syntactic consistency and conciseness*, can be achieved [15]. The absence of a concept mapping precludes the discovery of identifiers that violate the homonym restriction only. The approach is based on identifier *containment*: an identifier is *contained* within another if all of its soft words are present, in the same order, in the containing identifier. For example, the identifier position is contained in the identifier relative_position. The key assumption of the syntactic approach is that a maximal identifier – an identifier not contained in any other – is associated with a unique concept. Thus, the set of concepts is approximated using these *maximal* identifiers.

When an identifier is contained within another, one of two possible violations has occurred. One possibility, illustrated in Figure 3a, is that there is a single concept associated with the two identifiers; thus, violating Deißenböck and Pizka's synonym consistency requirement. For example, if the program contains only the concept *relative position* and there are the same two identifiers, position and relative_position, a synonym consistency violation exists. This is not a conciseness problem because position adequately describes the concept *relative position* when it is the only type of position used in the program.

The other possibility, is that the two identifiers map to different concepts in the concept space. In this case, the contained identifier violates Deißenböck and Pizka's conciseness requirement. An example of this situation is shown in Figure 3b where the identifier position maps to the concept *absolute position* and the identifier relative_position maps to the concept *relative position*. The identifier po-

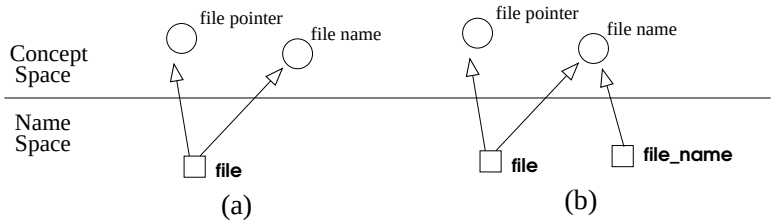

Figure 2. Illustration of the two types of violations. Figure (a) shows a homonym violation. Figure (b) shows how a synonym violation is also introduced by the function that opens a file.

sition violates the conciseness requirement by being too general a name for the concept *absolute position* because it might refer to a *relative position* as well.

4.2 Examples

The expansion can uncover a violation that was previously undetected and remove an errant violation. For example, without expansion the syntactic definition of conciseness and consistency cannot identify the violation between abspos and absolute_position. The expansion of sz to size is another example. In the other direction the expansion of dir to direction in one context and to directory in a different context removes the homonym violation in which dir was associated with the concepts *direction* and *directory*. This violation was previously undetectable using the syntactic approach.

The proceeding two cases involve identifiers composed of a single soft word. For identifiers composed of more than one soft word, the expansion can have the following four additional effects: it can cause an identifier to be newly contained in another, it can cause an identifier to newly contain another, it can remove a containee, and it can remove a contained identifier. As a representative example of these four (all four are similar), the first effect is seen in the expansion of main_buf to main_buffer. After expansion buffer is a newly contained soft word. Thus, pre-expansion, there is an undetected conciseness violation caused by main_buf and buffer potentially both referring to the concept of the *main buffer*.

Of the 2.6 million unique identifiers in all programs, 2.3% include a single soft word. From these 46,230 violations were uncovered and 25,195 simple errant violations were broken. At present, categorization for compound identifiers is not completely automated and thus cannot be easily quantified.

5 Related Work

Anquetil and Lethbridge (among others) have observed that there is some controversy over the value of general identifier names [1]. For example, Sneed finds that "in many legacy systems, procedures and data are named arbitrarily $\cdots$ programmers often choose to name procedures after their girlfriends or favorite sportsmen" [26]. A similar pattern was observed by one of the authors at a previous industrial position in the code of a colleague who was fond of Star Wars.

The work presented herein follows Anquetil and Lethbridge in assuming that software engineers are trying to give significant names (although they may have failed in this attempt) [1]. With the spread of true engineering discipline in the software construction process, this assumption grows increasingly more likely to be satisfied. Also following previous work (*e.g.*, that of Jones [11]), identifier quality is assumed to be correlated to the use of dictionary words and coherent abbreviations.

Work related to this topic comes from two different disciplines. Within the natural language processing community, work was been done on acronym expansion, some of which is highlighted below [14, 24]. Within the software engineering community, identifiers are receiving increasing attention both when finding characteristics of programs [1] and using them to better understand the domain of the program [5, 6, 18, 20, 12, 22]. The work highlighted here is somewhat different from the project discussed in this paper because up to this point, only rudimentary processing of identifiers has been attempted.

Acronym expansion has been studied within the context of written language. Larkey et al. developed a heuristic approach to the problem [14]. They mined web pages in search of acronyms and then relied on standard patterns of text to identify the correct expansion. This approach is not suitable to source code because of the reliance on textual patterns that do not occur in source code. Pakhomov worked on normalizing acronyms in medical text [24]. This

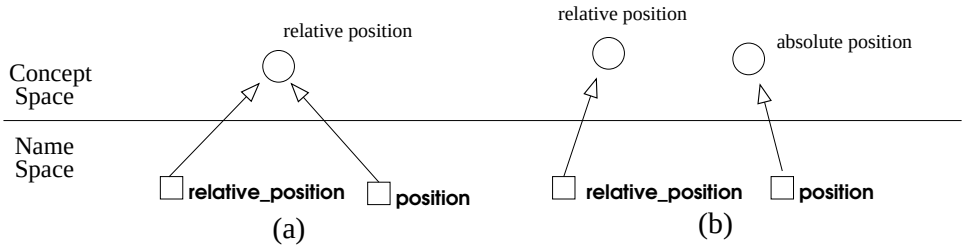

Figure 3. Shows the syntactic violations. Figure (a) shows a Conciseness Violation. Figure (b) shows a Synonym Consistency Violation.

has more similarity to the problem in source code because numerous abbreviations and acronyms are used throughout the text. The goal of this work is disambiguation (to decide which of several expansions is correct). A maximum entropy technique is used to decide which of the expansions is correct. The seven words before and after the abbreviation are used to determine the context. This technique may be applicable to choosing from among the correct expansions in source code and represents future work on the expansion algorithm.

Several researchers have made use of identifiers to find characteristics of programs. Anquetil and Lethbridge consider extracting information from type names in a large Pascal application [1]. In their definition two records implement the same *concept* if they have similar field names and types (though they are lax on enforcing type equivalence). Thus, this work provides a framework in which to study a form of concept identification (or at least concept equivalence) through types.

Another project considers file names rather then identifiers names. Following the work of Merlo et al. [23] who proposed clustering files according to the concepts referred to in the file's comments and the function names they contain, Anquetil and Lethbridge discuss techniques for extracting concepts (referred to as "abbreviations") from file names [2]. The task is difficult because file names rarely contain word markers (*e.g.*, capital letters, hyphens, and underscores). File names also tend to be short and include a large number of terse abbreviations.

Caprile and Tonella analyze function identifiers by considering their lexical, syntactical, and semantical structure [5]. They later present an approach for restructuring function names aimed at improving their meaningfulness [6]. The analysis involves breaking identifiers into well separated words (*i.e.*, hard words). The restructuring involves two steps. First, a lexicon is standardized by using only standard terms for composing words within identifiers. Second, the arrangement of standard terms into a sequence has to respect a grammar that conveys additional information. For example, the syntax of an indirect action, where the verb is implicit, is different from the syntax of a direct action. They were able to come up with an effective grammar for the restricted domain of function identifiers.

Finally, identifiers play a key role in several applications of information retrieval (IR) to software. For example, the early work of Maarek [18], which used IR to automatically construct software libraries, made heavy use of identifiers. More recently, Marcus et al. used IR to identify semantic similarities between source code documents [20]. Based on IR, similar high-level concepts (*e.g.*, abstract data types) are extracted as identified clusters in the code. In similar work, Kawaguchi et al. describe an automatic software categorization algorithm to help find similar software systems in software archives [12]. They explore several known approaches including code clones-based similarity metric, decision trees, and latent semantic analysis. Finally, in a related vein, Marcus et al. address the problem of concept location using latent semantic analysis [22]. Two concept locators are presented–one based on user queries and the other on partially automated queries.

6 Summary and Future Work

This paper has presented a first step toward the transformation of source code by expanding abbreviations (either permanently or as a 'tool tip' in a GUI). Such expansion can avoid mistakes made by programmers with an incorrect understanding of the abbreviations. It also helps subsequent analysis tools by making meaning easier to extract from the source code. Given the payoffs in terms of better comprehension and better utilization of identifiers in software tools, identifier expansion is a useful transformation.

Looking forward, additional work will consider abbreviations that appear in multiple locations. The key tradeoff here is whether or not to limit an abbreviation to a single expansion within a program. Furthermore, as noted in Section 3 better integration of the splitting and expansion process will improve the quality of the expanded identifiers. Fi-

nally, given that words in an identifier need to make sense as a unit, co-occurrence models can play a role. For example, two possible expansions of the identifier thenewestone are the_newest_one and then_ewe_stone. However, the soft words that make up the_newest_one have greater probability of co-occurring and thus are preferred over those of then_ewe_stone.

7 Acknowledgments

This work is supported by National Science Foundation grant CCR0305330. Special thanks to students of CS 302 for their participation in the evaluation.

References

[1] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, November 1998.

[2] Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names; a new file clustering criterion. In 20^{th} *IEEE International Conference and Software Engineering (ICSE 1998)*, pages 84–93, Kyoto, Japan, April 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10), October 2002.

[4] T. Brants and A. Franz. Web 1t 5-gram version 1, 2006. Linguistic Data Consortium, Philadelphia.

[5] B. Caprile and P. Tonella. Nomen est omen: analyzing the language of function identifiers. In *Working Conference on Reverse Engineering*, Altanta, Georgia, USA, October 1999.

[6] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM*, 2000.

[7] F. Deißenböck and M. Pizka. Concise and consistent naming. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*, St. Louis, MO, USA, May 2005. IEEE Computer Society.

[8] L. Etzkorn, L. Bowen, and C. Davis. An approach to program understanding by natural language understanding. *Natural Language Engineering*, 5(5), 1999.

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

[10] F. Feng and W.B. Croft. Probabalistic techniques for phrase extraction. *Information Process Management*, 37(2), March 2001.

[11] D. Jones. Memory for a short sequence of assignment statements. *C Vu*, 16(6), December 2004.

[12] S. Kawaguchi, P.K. Garg, M M. Matsushita, and K. Inoue. Automatic categorization algorithm for evolvable software archive. In *Proceedings of International Workshop on Principles of Software Evolution*, Helsinki, Finland, September 2003.

[13] R. Krovetz. Viewing morphology as an inference process. In R. Korfhage et al., editor, *Proceedings of the 16th ACM SIGIR Conference*, June 1993.

[14] L. Larkey, P. Ogilvie, M. Price, and B. Tamilio. Acrophile: An automated acronym extractor and server. In *Digital Libraries*, 200.

[15] D. Lawrie, D. Binkley, and H. Feild. Syntactic identifier conciseness and consistency. In *Proceedings of 2006 IEEE Workshop on Source Code Analysis and Manipulation (SCAM'06)*, Phidelphia, USA, September 2006.

[16] D. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *14th International Conference on Program Comprehension*, 2006.

[17] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th International Conference on Program Comprehension*, 2006.

[18] Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8), 1991.

[19] C. Manning and H. Schutze. *Foundations of statistical natural language processing*. The MIT Press, 1999.

[20] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *Proceedings of Automated Software Engineering*, San Diego, CA, November 2001.

[21] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the* 25^{th} *IEEE/ACM International Conference on Software Engineering*, Portland, OR, May 2003.

[22] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *IEEE Working Conference on Reverse Engineering*, Delft, The Netherlands, November 2004.

[23] E. Merlo, I. McAdam, and R. De Mori. Source code informal information analysis using connectionnist models. In *Ruzena Bajcsy, editor, IJCAI'93, International Joint Conference on Artificial Intelligence*, volume 2, Los Altos, Calif, 1993.

[24] S. Pakhomov. Semi-supervised maximum entropy based approach to acronym and abbreviation normalization in medical texts. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 2002.

[25] J. Rilling and T. Klemola. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *Proceedings of the* 11^{th} *IEEE International Workshop on Program Comprehension*, Portland, Oregon, USA, May 2003.

[26] H. Sneed. Object-oriented cobol recycling. In *3rd Working Conference on Reverse Engineering*. IEEE Computer Society., 1996.

[27] David A. Wheeler. SLOC count user's guide, 2005. http://www.dwheeler.com/sloccount/sloccount.html.