Syntactic Identifier Conciseness and Consistency

Dawn Lawrie
Loyola College
Baltimore MD
21210-2699, USA
lawrie@cs.loyola.edu

Henry Feild
Loyola College
Baltimore MD
21210-2699, USA
hfeild@cs.loyola.edu

David Binkley
Loyola College
Baltimore MD
21210-2699, USA
binkley@cs.loyola.edu

Abstract

Readers of programs have two main sources of domain information: identifier names and comments. It is therefore important for the identifier names (as well as comments) to communicate clearly the concepts that they are meant to represent. Deißenböck and Pizka recently introduced rules for concise and consistent variable naming. One requirement of their approach is an expert provided mapping from identifiers to concepts.

An approach for the concise and consistent naming of variables that does not require any additional information (e.g., a mapping) is presented. Using a pool of 48 million lines of code, experiments with the resulting syntactic rules for concise and consistent naming illustrate that violations of the syntactic pattern exist. Two case studies show that three quarters of the violations uncovered are "real". That is they would be identified using a concept mapping. Techniques for reducing the number of false positives are also presented. Finally, two related studies show that evolution does not introduce rule violations and that programmers tend to use a rather limited vocabulary.

Keywords

Identifier Quality, Part-of-speech

1. Introduction

Concise and consistent variable naming, as described by Deißenböck and Pizka, can improve code quality through improved identifier names [5]. The motivation for their work is the observation that "lousy naming in one place spoils comprehension in numerous other places," while the basis for their work is found in the quote "research on the cognitive processes of language and text understanding shows that it is the semantics inherent to words that determine the comprehension process" [5]. Other studies have also pointed to the importance of good identifier names. For example, Rilling and Kelmola observe "In computer programs, identifiers represent defined concepts" [11], while Caprile and Tonella point out that "Identifier names are one of the most important sources of information about program entities" [4].

Deißenböck and Pizka's technique, requires a mapping from the domain of identifiers to the domain of concepts. Such a mapping must be constructed by an expert. For new projects, this mapping can be constructed alongside the program with minimal addition cost. For existing programs, however, the cost can be prohibitively expensive. This paper considers *syntactic* concise and consistent naming. In other words, by only considering the syntactic makeup of identifiers, can a useful approximation to the techniques of Deißenböck and Pizka be achieved?

In more detail, Deißenböck and Pizka define three rules, one for *concise* and two for *consistent* identifier names. An identifier is *concise* if its semantics exactly match the semantics of the concept it is used to represent. For example, output_file_name concisely represents the concept of the name of an output file. (A related notion, *correctness* allows an identifier to represent a more general concept. For example, file_name correctly, but not concisely, represents the concept of the name of an output file, while the identifier foo neither correctly nor concisely represents the concept.)

There are two rules related to *consistent* identifiers. They identify inconsistencies caused by identifier homonyms and synonyms. In natural language, a homonym is one of two or more words spelled and pronounced alike but different in meaning (*e.g.*, 'waste' and 'waist') [14]. A synonym is one of two or more words or expressions of the same language that have the same or nearly the same meaning in some or all senses (*e.g.*, 'baby' and 'infant') [14].

In a program, an identifier is a *homonym* if it is associated with more than one concept from the program. For example, in a program dealing with both absolute and relative directory paths (two different concepts), the identifier path is a homonym as it is associated with more than one concept. As Deißenböck and Pizka emphasize, accurately knowing the set of all concepts used in a program is important. The identifier path is not a homonym in a program with only one path concept. Thus, it is important that only concepts from the program be considered. Otherwise, the concept space becomes too large and unwanted inconsistencies occur.

The second inconsistency involves *synonyms*: two identifiers that can be associated with a common concept. For example, the identifiers hash and hash_value are synonyms as each can represent the concept of an object's *hash value*. As a second example, the identifiers list_head and list_front are also synonyms (head and front are synonyms in English).

Identifiers that fail to be concise or consistent increase the comprehension complexity and its associated costs [5]. Such failures can be identified using Deißenböck and Pizka's techniques provided a mapping from identifiers to concepts is available. In the absence of such a mapping, it is still possible to identify a subset of these naming failures. Techniques for doing so are introduced and empirically investigated in this paper. In more detail, the primary contributions of this paper are the following:

1. The definition of syntax-based conciseness and consistency that does not require an expert-constructed mapping from identifiers to concepts.
2. An empirical investigation of a tool based on the syntactic definition. The experiments, which study almost 50 million lines of code, consider the prevalence of conciseness and consistency failures. Statistical models are used to better understand the collected data.
3. Two case studies. The first exhaustively considers all conciseness and consistency failures from two small programs. This study compares the tool's output to that produced by a human "oracle". The second case study considers a sampling of the conciseness and consistency failures from the larger program eMule, a 170 KLoC C++ program.
4. An empirical investigation of an observation by Antoniol et al., that programmers use a rather limited vocabulary [3, 2]. In particular, incorporating natural language synonyms does not dramatically increase conciseness and consistency failures.
5. Finally, a longitudinal study addresses the question "does evolution introduce conciseness and consistency failures?"

The rest of this paper first presents some necessary background material in Section 2. Definitions of syntactic conciseness and consistency are given in Section 3, followed by empirical studies in Section 4. Related work is then considered in Section 5. Finally, the paper concludes with a discussion of some topics for future investigation and a summary in Sections 6 and 7.

2. Background

This section provides context for the technique described in Section 3 and the empirical evidence presented in Section 4. It first, describes the WordNet tool used to obtain natural language synonyms and part of speech information and then the identifier splitting tool used to break identifiers up into their constituent parts. Information on the subject programs studied is then presented followed by information on the statistical tests used.

2.1 WordNet

WordNet is a lexical database of the English language developed by the Cognitive Science Laboratory at Princeton University [6]. WordNet contains 155,327 different words classified as nouns, verbs, adjectives, and adverbs. The power of WordNet comes from the relationships that have been identified, which includes topical relationships, synonym relationships, and homonym relationships to name a few. For example, WordNet identifies *head* and *mind* as synonyms. This ontology has been used by many researchers including those in the field of information retrieval [9] and data mining [8]. There is now an international conference devoted to WordNet which has reported on the creation of WordNet for other languages – Russian, Arabic, Persian, Korean, etc., and applications of the tool.

2.2 Identifier Splitting

Following others who study identifiers, identifiers are divided into their constituent parts for analysis [5, 4, 11, 2]. 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 demarked by the use of word markers (*e.g.*, the use of CamelCase or underscores). For example, the identifiers sponge_bob and spongeBob both contain the well separated hard words sponge and bob.

For many identifiers, the division into hard words is sufficient. This occurs when all the hard words are dictionary words or known abbreviations. When the hard word is neither, the identifier contains non-well-separated words. The identification of these "*soft words*" is the goal of identifier splitting. For example, the identifier zeroindeg includes a single hard word because there are no word markers; thus, division into hard words is insufficient. The splitter divides this hard word into the three soft words zero-in-deg (*i.e.*, zero, in, and deg, a known abbreviation). The algorithm does this by using a greedy approach that recursively compares the longest prefix and suffix that are in the dictionary or a known abbreviation list.

2.3 Subject Programs

The analysis includes empirical data collected from 186 programs. Up to 70 versions of a program were considered to support the longitudinal studies. Ignoring multiple versions, 78 unique programs were considered. All but 12 were open source programs. Programs ranged in size from 1,423 to 3,087,545 LoC and covered a range of application domains (*e.g.*, aerospace, accounting, operating systems, program environments, movie editing, games, etc.) and styles (comment lines, GUI, real-time, embedded, etc.). Most of the code was written in C. Significant C++ and Java code were also studied along with a small amount of 30 year old Fortran code. Several of the programs were written by programmers whose native language was not English. For these programs the analysis was performed using a dictionary for the programmer's native language or, if multiple languages were evident in the code, the union of the respective dictionaries. (The publicly available dictionaries that accompany the Linux spell checker ispell version 3.1.20 were used.)

Figure 1 shows 10 representative C, C++, and Java subject programs. The two Fortran programs are not shown in the figure. They are PLM compilers from 1975 and 1981 and include 9,704 and 11,478 LoC, respectively.

The figure reports code sizes for C, C++, and Java (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 [15], 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. The last two columns present the start year of the project and its release year. These dates were extracted from program documentation (internal and external). In general, the release year is more accurate as in can be difficult to determine the start year for a program that includes third party libraries written before the program "started".

program	wc				sloc	year	
	C	C++	Java	Total	Total	start	release
cinelerra-2.0	1,044,996	106,357	0	1,151,353	820,980	1996	2004
cpm68k1-v1.2a	132,171	0	0	132,171	102,252	1978	1984
empire_server	85,548	0	0	85,548	62,793	1985	1998
eMule0.46c	1,759	172,164	0	173,923	135,567	1999	2005
I4.2	2,109,050	398,463	502,965	3,010,478	1,704,823	1993	2004
jakarta-tomcat-5.5	68,003	0	353,604	421,607	219,766	1999	2005
LEDA-3.0	41,610	0	0	41,610	27,425	1988	1992
minux-2.0	326,210	0	0	326,210	244,033	1980	1996
mozilla-1.4	1,047,741	1,949,292	6,493	3,003,526	2,107,436	1998	2003
quake3-1.32b	353,806	57,431	0	411,237	281,432	1999	2005
Totals for all code not just that shown above							
open source	19,170,546	14,587,482	6,327,380	40,106,590	27,129,263		
proprietary source	7,167,689	787,094	582,107	8,536,890	5,391,815		
all	26,338,235	15,374,576	6,909,487	48,643,480	32,521,078		

Figure 1. Subject Programs (proprietary programs are named I#).

Figure 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 top 14 rows of the table present a representative sample of the programs. The bottom seven rows summarize the data over all programs (not just that of the representative programs in the top of the table). Summaries include two orthogonal groupings (open source versus proprietary, and by programming language) and all the data taken collectively.

2.4 Statistical Tests

Several statistical techniques are used in the interpretation of the data gathered during the study. This section introduces these techniques and can be skipped by those familiar with statistics.

When a simple linear association is of interest, Pearson's linear regression model, which measures linear correlations between variables, is built. For the effect of explanatory variables X, Y, and Z on response variable A, the resulting model coefficients, m_i, belong to the linear equation $A = m_1X + m_2Y + m_3Z + b$. Each coefficient has an associated p-value. A p-value less than 0.05 represents a significant explanatory variable.

For more complex models, linear mixed-effects regression models [13] are used to analyze the data. Such models allow the examination of important effects that are associated with the response variables. The initial model includes explanatory variables and a number of interaction terms. The interaction terms allow the effects of one variable on the response variable to change depending upon the value of another variable. Backward elimination of statistically non-significant terms ($p > 0.05$) yields the final model. Note that some non-significant variables and interactions are retained to preserve a hierarchical well-formulated model [10].

In these models, Tukey's highly significant difference method for multiple comparisons is used. However, computing a standard t-value for each comparison and then using the standard critical value increases the overall probability of a Type I error. Thus, Bonferroni's correction is made to the p-values to account for multiple comparisons. In essence each p-value is multiplied by the number of comparisons and the adjusted p-value is compared to the standard significance level (0.05) to determine significance. Tukey's method and Bonferroni's correction were chosen because they are both rather conservative tests.

With both Pearson's test and the linear mixed-effects regression models, the coefficient of determination, R^2, is reported. This coefficient is interpreted as the proportion of the variability of the response variable that is explained by the selected explanatory variables. This coefficient ranges from 0 to 1; the closer to 1, the better the model.

3. Technique

Deißenböck and Pizka describe a formal model for adequate identifier naming that includes rules for the *correct* and *concise* naming of identifiers [5]. Their rules makes use of the set of all concepts relevant to a program and provide "a formal model based on bi-jective mappings between concepts and names."

The rules include three requirements involving homonyms, synonyms, and conciseness. These three can be 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 3a). 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 3b). Finally, an identifier i for concept c is concise provided no concept less general than c is represented by another identifier (an example is given in the next paragraph).

Deißenböck and Pizka present a case study in which maintenance introduces the seven identifiers pos, apos, abspos, relpos, absolute_position, relative_position, and position representing two concepts c_1 = "*absolute position*" and c_2 = "*relative position*". The identifier position fails the homonym consistency requirement as it is associated with more than one concept from the program (in this case concepts c_1 and c_2). In addition, the study determined that relpos and relative_position were both used for concept c_2, which violates the synonym consistency requirement. Finally, the identifier position would concisely represents the concept absolute_position provided that the program did not include

program	dominant language	start year	release year	LoC (wc)	unique ids	id instances	hard words	soft words	percent[‡] increase
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%
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%
I1	C	1987	1997	454,609	30,092	482,228	48,125	82,307	71.0%
I4.2	C	1993	2004	3,010,478	113,662	2,694,901	328,079	422,364	28.7%
I6.6	C	2000	2002	237,257	10,791	104,290	29,207	34,549	18.3%
jakarta-tomcat-5.5.11	Java	1999	2005	421,607	19,202	351,487	48,537	54,471	12.2%
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%
plm80s	Fortran	1975	1977	9,704	581	22,314	581	886	52.5%
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%
Totals for (over all code not just shown)	instances per id	hard words per id	soft words per id	LoC (wc)	unique ids	id instances	hard words	soft words	percent increase
open source	19.2	2.7	3.2	40,106,590	2,504,937	48,098,029	6,817,779	8,040,625	17.9%
proprietary	19.6	2.7	3.5	8,536,890	385,792	7,543,663	1,055,329	1,331,327	26.2%
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%
Fortran	18.0	1.4	1.8	21,182	2,238	40,273	3,141	3,993	27.1%
All	19.3	2.7	3.2	48,643,480	2,890,153	55,638,621	7,872,119	9.370,562	19.0%

Figure 2. Basic counts from 14 selected programs. Some of the programs from Figure 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.

any other position concepts (*e.g.*, *relative position*). As the program included multiple specific kinds of positions, the identifier position fails the conciseness requirement.

In most instances, when the homonym requirement is violated the synonym requirement is also violated. Figure 3 illustrates this. The identifier file is a homonym associated, in different parts of the program, with the concept of a *file name* and elsewhere a *file pointer*. If the two concepts are to be referred to in the same scope (at least in a strongly typed language) then at least one additional identifier would be required as shown in Figure 3b. However, the inclusion of this second identifier introduces a synonym violation as the identifiers file and file_name both refer to the same concept.

In this example any function that opens a file would need to refer to both the *file name* and *file pointer* concepts. As an example in which it is plausible that a homonym would exist in the absence of a synonym, consider the situation shown in Figure 3c in the context of a program that reads a directory path into the variable path and then passes it to either function f1 or f2 depending on the path being relative or absolute. If f1 and f2 use the name path for their formal parameter, then the program includes two concepts *relative path* and *absolute path* and only uses one identifier, path, to refer to them. This violates the homonym rule, but not the synonym rule.

The absence of a concept mapping precludes the discovery of identifiers that violate the homonym restriction only. Testing that identifiers satisfy a restricted form of synonym consistency and conciseness, can be achieved *syntactically* (*i.e.*, without the identifier to concept mapping). It turns out that a similar pattern indicates a violation of the syntactic-synonym consistency requirement and the syntactic conciseness requirement. Both involve an identifier being *contained* in another. Here containment results when one identifier includes, in the same order, all the soft words from another. For example, the identifier relative_position includes two hard words each composed of a single soft word. Thus this identifiers includes, in order, all the soft words from the identifier position.

An important implication of this containment is that the concepts associated with the two identifiers have a non-empty intersection; thus, violating Deißenböck and Pizka concept space by definition. The presence of a second containing identifier (*e.g.*, absolute_position), which also contains position, implies Deißenböck and Pizka rule for conciseness has also been violated. It does so because the two containing identifiers imply the program includes two separate concepts, but the contained identifier does not precisely represent either of them. More formally, the two violations are defined as follows.

Definition 3 Syntactic Synonym Consistency and Conciseness. Let identifier id_1 be the sequence of soft words $sw_1\,sw_2\,\cdots\,sw_{n1}$. Identifiers id_1 and id_2 fail the *syntactic synonym consistence requirement* if id_2 includes the sequence of soft words $w_1\,w_2\,\cdots\,sw_1\,sw_2\,\cdots\,sw_{n1}\,\cdots\,w_{n2}$ (*i.e.*, $id_2 = w_1\,w_2\,\cdots\,id_1\,\cdots\,w_{n2}$). Furthermore, id_1 fails the *syntactic conciseness requirement* if there exists a third identifier id_3 that includes the sequence of soft words $u_1\,u_2\,\cdots\,sw_1\,sw_2\,\cdots\,sw_{n1}\,\cdots\,u_{n3}$.

Section 4 empirically investigates two important questions related to this definition. First, do synonym consistency and conciseness failures exist in real code? Obviously, the technique is of little interest if violations are infrequent or non-existent. Second, are syntactic violations indicative of violations using the Deißenböck and Pizka concept-map based definitions? If the syntactic ap-

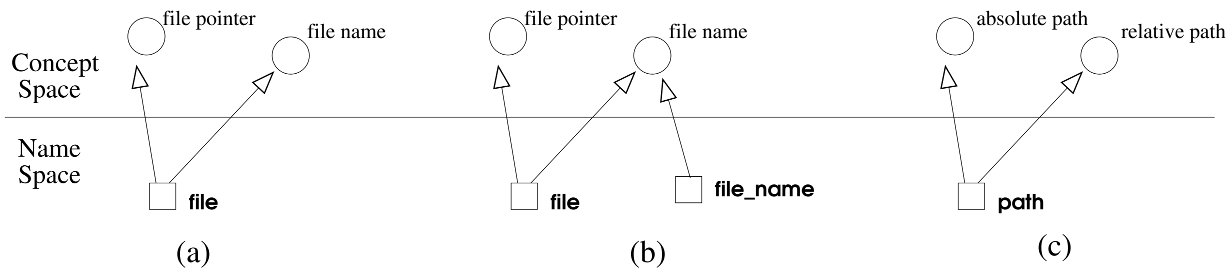

Figure 3. Illustration of the two types of syntactic violation. Figure (a) shows a homonym violation. Figure (b) shows how a synonym violation is also introduced by the function that opens a file. Finally, Figure (c) shows a plausible homonym only example.

proach can identify a useful subset of the violations, without the need for a concept mapping, then it forms the core of a useful tool.

Section 4 also investigates a related hypothesis suggested by the following observation of Antoniol et al., "Programmers tend to process application-domain knowledge in a consistent way when writing code: program item names of different code regions related to a given text document are likely to be, if not the same, at least very similar" [3, 2]. This observation suggests programmers use a limited vocabulary and can be tested using WordNet to identify all possible synonyms for each soft word in an identifier. More formally, assume that for soft word w, $S(w)$ denotes the natural language synonyms of w. In Definition 3 replace $w_1\, w_2\, \cdots\, sw_1\, sw_2\, \cdots\, sw_n\, \cdots\, w_m$ with $w_1\, w_2\, \cdots\, s_1\, s_2\, \cdots\, s_n\, \cdots\, w_m$, where $s_i \in S(sw_i)$ and the corresponding replacement for id_3. For example, using WordNet, the identifiers list_head and list_front violate the synonym rule as *head* and *front* are natural (English) language synonyms.

This section concludes by considering one of several refinements to syntactic conciseness and consistency. Section 6 (future work) describes others. Definition 3 is a straightforward restriction of Deißenböck and Pizka's work in the absence of an identifier to concept mapping. It is possible to improve upon this by exploiting certain grammatical patterns that indicate different concepts.

For example, one common pattern seen in the empirical studies is to have two identifiers where one is a noun phrase and another that includes a verb with this noun phrase. For example, tree_node and visit_tree_node. Syntactically, tree_node is contained in visit_tree_node and thus a (syntactic) violation. However, these two identifiers are associated with different (related) concepts and thus no violation exists in the Deißenböck and Pizka sense. Using WordNet to identify parts of speech, this pattern is easy to detect. Section 4 empirically investigates the frequency of this pattern.

4. The Study

Data regarding syntactic consistency and conciseness failures found in the 186 programs is presented through five empirical studies. The first summarizes statistics over all programs. Next, two case studies are considered, one exhaustive and one sampling. Finally, a longitudinal study and an investigation of the impact of incorporating natural language synonyms into identifiers are considered.

4.1 Statistics over all Programs

The ability to identify syntactic conciseness and consistency failures is of little value if the pattern does not occur in practice. Figure 4 shows the percent failure for 42 representative programs along with the number of unique identifiers in each program and the percentage of *severe* failures in which the contained identifier includes at least three soft words. The chart below the table, shows the failure percents for all 186 programs. Synonym and conciseness failures are sorted independently; thus, vertical comparisons do not reflect a particular program. The shape of the curves provides a general feel for the distribution of the data. Based on the last row in the table, an average program includes just over 2900 identifiers that fail the synonym requirement and just over 1300 that fail the conciseness requirement. This indicates that sufficient violations exist in practice to warrant further study.

From the table in Figure 4, it appears that the percentage of synonym failures is not strongly correlated to the number of unique identifiers. This is statistically true (R^2 = 0.12). To better model the percentage of synonym failures, backward elimination starting with the explanatory variables program size, start year, release year, programming language, and open source was used. The resulting model includes only release year and programming language. Notably absent is any measure of program size. The model's R^2 value of 0.48 means that it explains just under half of the variation in the percentage. This model indicates an increases of 0.42% synonym failures for each year later a project is released and a 4.3% increase for Java programs (no other language made a significant difference).

The model for the percentage of conciseness failures is less informative and more complicated. Its R^2 value of 0.22 indicates that less than a quarter of the variation in the percentage of conciseness failures is explained by the model. The final model includes the following explanatory variables: start and release years, program language, and open-source. The main complication in this model is an interaction between open source and release year. Thus release year has a different effect on open and proprietary source code. In this case, an increase in release year brings an increase of 0.14% to the percentage of conciseness failures in open source code while it brings a reduction of 0.28% in the percentage of conciseness failures in proprietary code. Neither of these percentages are large. In addition, every year later a project was started it has 0.12% fewer failures and, as with the synonym failures, Java brings a greater percentage. In this case 1.9% more conciseness failures.

program	unique identifiers	Failures		Severe Failures	
		Synonym	Conciseness	Synonym	Conciseness
LEDA-2.1.1	2226	21%	10%	2%	2%
LEDA-3.1.2	2946	20%	8%	1%	1%
a2ps-4.12	3593	22%	10%	3%	2%
apache_1.3.29	8040	19%	8%	4%	3%
barcode-0.98	344	21%	6%	5%	1%
*byacc.1.9	507	20%	9%	1%	0%
cinelerra-2.0	71995	21%	9%	6%	4%
*compress	164	11%	5%	0%	0%
cpm68k1-v1.3	2417	12%	6%	0%	0%
cvs-1.11.1p1	5552	20%	9%	3%	2%
eMule0.46c	21372	17%	8%	6%	4%
eclipse-2.1	83207	26%	11%	10%	6%
eclipse-3.2m4	155932	25%	9%	11%	4%
*genesis-all-3.0	2110	33%	12%	13%	5%
ghostscript-7.07	26546	19%	9%	6%	4%
gnuchess-4.0	1198	16%	8%	1%	1%
*gnugo-1.2	114	15%	8%	0%	0%
gnugo-2.0	627	15%	6%	1%	1%
gnugo-3.0.0	3118	21%	9%	3%	2%
httpd-2.0.48	16975	19%	9%	5%	3%
I1	29619	17%	12%	5%	4%
*I4.1	92547	21%	11%	10%	7%
I4.2	110727	21%	11%	10%	7%
I6.1	9869	16%	8%	5%	4%
I6.6	10583	16%	8%	5%	4%
I9	41189	20%	9%	7%	5%
I12	1098	19%	11%	4%	3%
jakarta-tomcat-3.0	3920	24%	9%	5%	3%
jakarta-tomcat-5.5	18416	25%	10%	7%	4%
javabb_073	1716	27%	9%	5%	2%
minux-2.0	21076	15%	7%	1%	1%
mozilla-1.0	173124	22%	9%	8%	5%
mozilla-1.6	176318	22%	9%	8%	5%
mysql-5.0.17	46297	21%	9%	7%	4%
*pacifi3d0.3	1139	11%	5%	1%	1%
*plm80s	539	8%	4%	0%	0%
quake3-1.32b	28676	18%	8%	5%	3%
samba-3.0.0	22553	20%	9%	8%	4%
spice3f4	9845	18%	10%	5%	4%
*tile-forth-2.1	661	34%	22%	2%	2%
uupc	147	12%	7%	1%	1%
Min	114	8%	4%	0%	0%
Max	181032	34%	22%	13%	7%
Average	14512	20%	9%	4%	3%

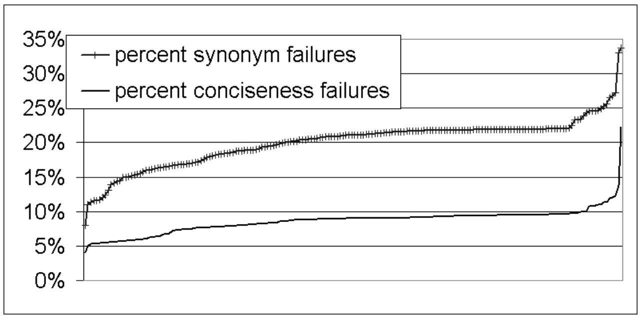

Figure 4. Percent synonym-consistency and conciseness failures. (A "*" marks programs with a minimum or a maximum value. Proprietary programs are named I#.)

4.2 Exhaustive Case Studies

Given that a significant number of syntactic synonym and conciseness violations occur, the next question to address is "are these violations real?" There are two possible differences between the violations that the syntactic approach reports and those obtained using a concept mapping. Clearly the syntactic approach will miss violations when the identifiers do not share common source words. For example, the identifiers file, fp, and fin might be synonyms (all representing the *file pointer* concept), but syntactic approach cannot determine this.

The other difference involves identifiers for which the syntactic approach identifies a violation, but no violation exists when using the associated concepts. To determine how many such false positives the syntactic approach produces, hand inspection of all violations from two of the smaller programs was performed. As shown in the following table, this inspection produced five categories. The encouraging news is that 72% of the synonym violations and 76% of the conciseness violations were true violations. For example, the identifiers status and file_status violate the synonym consistence requirement while the identifiers home_dir and in_home indicate two refinements of the concept *home*, which means the identifier home fails the syntactic conciseness requirement.

Description	Synonym		Conciseness	
(1) violation	49	72%	22	76%
(2) non violations	6	9%	3	10%
(3) attribute	9	13%	1	3%
(4) verb-noun phrase	3	4%	3	10%
(5) struct field	1	1%	0	0%
All	68	100%	29	100%

Many of the remaining identifiers were non-violations (9% of the synonym violations and 10% of the conciseness violation). For example, prefix is contained in isolate_tilde_prefix. While prefix could be replaced with string_prefix in the string concatenation routing where it is found, isolate_tilde_prefix is a function whose associated concept does not overlap with that of prefix. A conciseness example from uucp is the identifiers FILE, copy_file, file_mode, and log_file. As FILE is a type, it's concept is separate from the others, although the syntactic algorithm cannot, at present, make this determination.

The remaining three categories all suggest refinements to the technique. The third category includes what Ada refers to as attributes and C# as properties. For example, the two identifiers cwd and cwd_len (a synonym failure), and the three identifiers result, result_index, and result_size (a conciseness failure) include *variable properties*. Here, by convention, programmers recognize identifiers such as cwd and result as the underlying value of which the other identifiers are properties.

The fourth category was mentioned at the end of Section 3. An example includes the identifiers home_dir and get_home_dir, which violate the syntactic synonym rule, but are associated with different concepts. Using part-of-speech information, this case can be identified when two identifiers differ by a verb. This pattern is explored further in Section 4.5.

The final category includes a structure field adr and the local identifier next_adr. Deißenböck and Pizka do not explicitly discuss structure fields, but including the structure name (letter in the case), seems a straight forward extension of their work that removes the synonym failure in this example.

4.3 eMule Case Study

The case study from the previous section considers all the violations in two small programs. This section presents a "selective case study" of eMule a 170 KLoC C++ program chosen at random from the larger programs. Examining eMule's 3725 synonym failures and 1762 conciseness failures is prohibitively expensive. Instead seven representative examples were selected. Each includes three parts: the base (contained) identifier, the identifiers that contain it, and a discussion.

(1) m_strHost (the contained identifier)
m_strHostName

The first case is the classic synonym violation in which a concept that already has a name receives another. In this case, the identifier m_strHost and the identifier m_strHostName both denote to the same concept (the string representation of the host computer to connect to).

(2) CheckDiskspace
CheckDiskspaceTimed

As a second classic example, eMule includes two methods for checking if sufficient disk space exists to write a file. Their names, CheckDiskspace and CheckDiskspaceTimed, clearly fail to satisfy Deißenböck and Pizka's definition of consistency as they both refer to the concept of a *timed disk check*. In this instance, one obvious fix would be to rename the first method CheckDiskspaceUntimed or something similar. This would disambiguate the names for the two concepts of timed and un-timed disk space checks.

(3) IcmpCloseHandle
lpfnIcmpCloseHandle

The third example illustrates a case in which synonym restriction is formally violated, but knowing a little about the identifiers removes any real issue. EMule includes the class type IcmpCloseHandle and the variable lpfnIcmpCloseHandle of that type. Both identifiers represent the same concept, but knowing that one is a type name disambiguates the two.

(4) m_n_file
m_n_file_size

The identifiers m_n_file and m_n_file_size form a less egregious synonym violation. The method "int CZIPFile::GetCount() { return m_nFile; }" suggests that consistency could be attained by renaming m_nFile to m_nFileCount

(5) m_wndSplitter
m_wndSplitterchat
m_wndSplitterirc
m_wndSplitterstat
...

The eMule class CSplitterControl implements a window splitter control. The server window includes a window splitter, under the name m_wndSplitter, as do several other windows. For example, the "chat" window includes m_wndSplitterchat which, like m_wndSplitter is of type CSplitterControl. (Note that this identifier is not well separated and thus identifier splitting into soft words is required to uncover this conciseness failure.) It is hard to know if the program's evolution began with a single splitter (in the server class) and the others were subsequently introduced or not, but in order to have concise names, m_wndSplitter should be renamed m_wndSplitterServer.

(6) GetFileType
GetFileTypeDisplayStr
GetFileTypeByName
GetFileTypeSystemImageIdx
GetFileTypeDisplayStrFromED2KFileType

The penultimate example involves five identifiers. The existence of the second, indicates a synonym violation and means that the first needs to be replaced to separate its concept from that of getting a displayable string representation for a file type. One naive way of doing so is to replace the first identifier with GetFileTypeNonDisplayStr or GetFileTypeInternalStr. Note that the latter of these conflicts with the third identifier.

A snippet showing the definition of the third identifier appears in Figure 5(a). As is clear from the comments proceeding the definition, to achieve conciseness, the third identifier should be replacement with something like GetFileTypeInternalByName. Similarly, to achieve conciseness, with the fourth identifier, the first would need to be separated from the concept of an "image index".

Finally, part of the definition of the fifth identifier is shown in Figure 5(b). Here the comment proceeding the definition muddies the water as the method produces an internal file type, but unlike GetFileTypeByName, this one appears to be appropriate for the GUI. This implies that internal file type names can be suitable for the GUI or not. Something the names of the two methods fail to make clear. For example, the identifier GetFileTypeByName should bear more in common with GetFileTypeDisplayStrFromED2KFileType as it too returns an internal, type name. As with the others, this identifier also conflicts with the first. The name for GetFileType would need to take all these concepts into account. To the extent that this example seems confusing, it is an excellent indication of the value of concise and consistent identifiers, as they would have had helped make clear the various concepts related to type names.

(7)

ident
IPHeader.ident (a field)
m_bLogSecureIdent
m_htiLogSecureIdent
...

```
// Return file type as used internally by eMule,
// examining the extension of the given filename
CString GetFileTypeByName(LPCTSTR pszFileName)
{
    ...
}
```

(a)

```
// Returns a file type which is used eMule internally only (GUI)
CString GetFileTypeDisplayStrFromED2KFileType(LPCTSTR pszED2KFileType)
{
  ...
}
```

(b)

Figure 5. Code snippets for the conciseness case study.

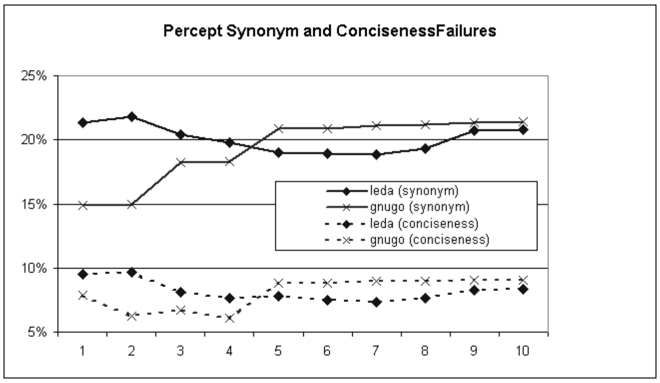

Figure 6. Two example programs from the longitudinal study.

The final example, is really a non-example. The identifier ident, which is contained in 37 other identifiers, exists in two separate contexts. First, it is a local variable of the method CIrcMain::Connect(). As there is no real conflict with the associated concepts for this local variable, it suggests that scope information might play a role in suggesting to an engineer when a violation might be a false positive. The second use of ident is as a field of the structure IPHeader. One might view it's *full* name as IPHeader.ident which would be a more concise name. Deißenböck and Pizka do not discuss using context provided by a scope or a type (class or structure), but it seems a straight forward improvement.

4.4 Longitudinal Study

Does evolution introduce synonyms? In principal, as a program ages, if it takes on new concepts then identifiers that were previously consistent and concise may become inconsistent and "un-concise". This occurs when software evolution introduces new identifiers (and their associated concepts). For example, the program which, actually the getopt library, originally only processed the short-form command line options. Later, a long form was added. The current code includes the identifiers options and long_options. Knowing the code's history, options is understood to be associated with the concept of *short options*. While options was originally a consistent identifier, the introduction of the concept of *long options*, means that it is no longer consistent. As as second example, position concisely represents the concept absolute_position provided that the program does not include any other kind of position (*e.g.*, relative_position).

Seven of the programs studied included four or more versions. Statistically, modeling the percentage of synonym and conciseness failures as a function of the version number, there is essentially no evidence that evolution introduces synonyms. This is visually apparent with the two examples shown in Figure 6. Leda is typical of most of the programs showing some ups and downs but no significant trend. Gnugo shows a slight increase early, but then levels out and remaining flat from versions 10 through 70.

4.5 WordNet

As introduced in Section 2, WordNet is a powerful tool for processing natural language. Identifiers are often composed of dictionary words, and thus WordNet can aid in their analysis. Two applications of WordNet are considered in the section. The first examines the breadth of the vocabulary used by programmers and the second considers how certain false positives can be identified using part of speech information.

When inspecting the identifiers in source code several authors (*e.g.*, Antoniol et al. [2] and Caprile and Tonella [3]) essentially observed that programmers use a limited vocabulary. For example, free can be an adjective, a verb, or adverb, but Caprile and Tonella discovered that it was only ever used as a verb. One implication of this is that similar concepts are given similar names. To formally investigate this observation, the consistencies and conciseness failures in all programs were recomputed after factoring in natural language synonyms. Doing so allows the tool to correctly determine that the identifiers list_head and list_front are synonyms as head and front are natural language synonyms in English. Finding very few additional violations, this experiment supports the observation the programmers use a limited vocabulary.

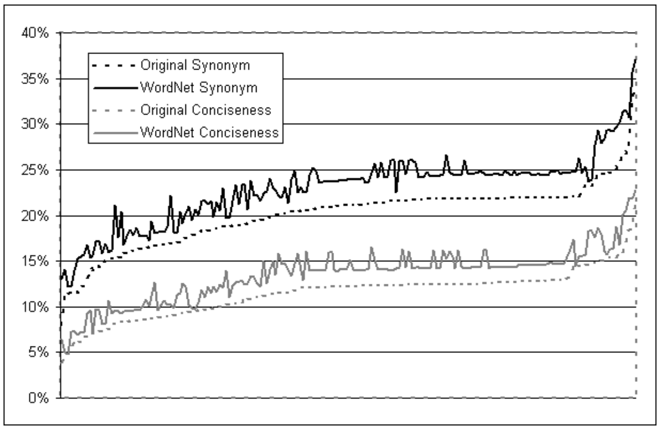

Figure 7. Incorproating natural language symonyms from WordNet. The x-axis shows each prorgam sorted seperately for synonym and concisness violations without using WordNet.

Statistically, for synonym violations, incorporating WordNet increases the number of violations 2.8% (R^2 = 0.998). This is shown graphically by the two black lines of Figure 7. The black jagged (solid) line shows the impact of using WordNet as compared to the non-WordNet data shown by the dashed line. For conciseness violations the increase is only 2.1% (R^2 = 0.996). This is shown graphically by the two lower gray lines. The rather minimal increase for both synonym and conciseness violations supports the observation that programmers use a limited vocabulary.

The second use of WordNet is to categorize identifiers based on certain grammatical patterns. This is, in essence, the start of a grammar-based technique similar to function-name grammar of Caprile and Tonella [4]. Two patterns were used in this preliminary study. Both are based on studies of the tool's output. They match identifiers that include a noun-phrase and a single additional soft word that is either a verb or an adjective. This additional soft word may come before or after the noun phrase. The verb form comes from functions that act upon data (the noun phrase). Examples, found by the tool, are shown in the top of Figure 8. Here the verb typically comes before the noun phrase. The adjective form comes from variables that represent attributes of other variables. Examples, found by the tool, are shown in the bottom of Figure 8. Again, the adjective typically comes before the noun phrase.

The verb-noun_phrase analysis is conservative in that only words that are exclusively used as verbs are considered. For example, consider the identifiers edit_clip and free_node. The word "*edit*" only appears as a verb in English, while the word "*free*" can also be used as a noun, adjective, and adverb; thus, edit_clip was counted, but free_node was not. Similarly, the adjective-noun_phrase form required words that only occur as adjectives.

Numerically, the verb form accounts 4.5% of the synonym violations. This is consistent with the percentage identified in the exhaustive case study of Section 4.2. The adjective form accounts for 2.2%, or about half as many of the violations. Together the two grammar based patterns identifier 6.7% of the violations. Assuming that the case study from Section 4.2 is representative, this represents about one quarter of the false positives.

Verb-noun_phrase	
absolute_path	get_absolute_path
birth_day	get_birth_day
base_name	parse_base_name
user_name	send_user_name
arena	unlock_arena
clip	clip_edit
Adjective-noun_phrase	
background_color	background_color_selected
bit	highest_bit
history	previous_history
token	preceding_token
tokens	saved_tokens
child	previous_child

Figure 8. Grammar examples.

5. Related Work

This section considers four related projects that focus on identifier names. First, Anquetil and Lethbridge consider extracting information from type names in a large Pascal application [1]. They define two records to 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.

Taking type information into account is an example of the kind of information that a fact extractor (*e.g.*, Columbus [7]) can extract about identifiers. For example, tree_node is contained in visit_tree_node, and position is contained in absolute_position. Knowing that visit_tree_node is a function and tree_node a formal parameter of the function indicates that the two are associated with different concepts and thus not a violation of the synonym rule in the same way that two global integer variables position and absolute_position are.

Caprile and Tonella analyze function identifiers by considering their lexical, syntactical, and semantical structure [3]. They later present an approach for restructuring function names aimed at improving their meaningfulness [4]. 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 as 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. Extending this to all identifiers is a non-trivial task, but the resulting grammar would be useful in refining the notion of syntactic consistency and conciseness.

Deißenböck and Pizka stress the value of identifiers in source code [5] as they make up a significant amount of the unique information available from the source. For example, Eclipse 3.0M7 has 94,829 different identifiers which is around the same number of words as in Oxford Advanced Learner's Dictionary. They also introduce a tool that enforces the rules for consistent and concise identifiers during program construction. This is done with the aid of an identifier dictionary. The tool improves the productivity of programmers.

Finally, Takang et al. note that there is some controversy on the value of dictionary word identifiers [12]. For example, Shneiderman and Mayer report that "variable names had a statistical significance on comprehension." However, their study included only beginning students as participants. On the flip side, Sheppard et al. observe that "variable names did not have a statistical significance on the subject's performance." This was based on an experiment that involved 36 professional programmers. In this second experiment, the programs were quite small (they varied between 26 to 57 lines of code), which may have been too short to bring out differences especially with professional programmers.

6. Future Challenges

The current tool does not discover the violation that occurs between absolute_path_given and abs_path because abs is an abbreviation of absolute. Definition 3 could be broadened to include such cases as follows: for soft word w, let $A(w)$ denote the set of all dictionary words appearing in the program that map to the same concept as w. In Definition 3 replace $w_1\, w_2\, \cdots\, sw_1\, sw_2\, \cdots\, sw_n\, \cdots\, w_m$ with $w_1\, w_2\, \cdots\, a_1\, a_2\, \cdots\, a_n\, \cdots\, w_m$, where $a_i \in A(sw_i)$ and the corresponding replacement for id_3. As absolute is in $A(\text{abs})$ the above violation would be detected.

While presently unimplemented, the abbreviation relation, A, could be approximated by performing a wild-card search in the the documentation (both internal and external). For example, the search for "a.b.s." where "." represents any sequence of valid identifier characters in the mozilla source yields a single dictionary word, absolute. Two other examples occurring in the case study were extracted with the help of the unix utilities grep and ispell include horiz abbreviating *horizontal* and triag abbreviating one of the words *triangle* or *triangulate*.

Finally, in generating the examples used in the case studies it became clear that following the rules produced improved code that was easier to comprehend. However, this is an ideal that may be difficult to reach. For example, consider trying to motivate replacing buf with buf_value to avoid a conflict with the identifier buf_len. By convention, most programmers would understand that buf referred to the buffer's value. Empirical evidence as to the impact of allowing such "violations" on programmer comprehension is another area of future investigation.

7. Summary

Deißenböck and Pizka's propose the enforcement of rules for consistent and concise identifiers using a tool that incrementally builds and maintains an identifier dictionary as a system is being developed. The identifier dictionary "explains the language used in the software system, aids in consistent naming, and improves productivity of programmers by proposing suitable names depending on the current context." [5]. This paper studies the restriction and extension of Deißenböck and Pizka's rules that is computable without a mapping from names to concepts. As the empirical evidence shows, these syntactic rules are useful in identifying consistent and conciseness identifiers.

8. Acknowledgments

This work is supported by National Science Foundation grant CCR0305330.

9. 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] 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.

[3] B. Caprile and P. Tonella. Nomen est omen: analyzing the language of function identifiers. In *Working Conference on Reverse Engineering*, pages 112–122, Altanta, Georgia, USA, October 1999.

[4] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM*, pages 97–107, 2000.

[5] 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.

[6] C. Fellbaum, editor. *WordNet – An Electronic Lexical Database*. MIT press, 1998.

[7] R. Ferenc, ?. Besz?des, M. Tarkiainen, and T. Gyim?thy. Columbus - reverse engineering tool and schema for c++. In *IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 3–6, Montreal, Canada, October 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[8] I. Jonyer, D.J. Cook, and L.B. Holder. Graph-based hierarchical conceptual clustering. *Machine Learning Research Archive*, 2:19 – 43, March 2002.

[9] R Mandala, T Takenobu, and T Hozumi. The use of wordnet in information retrieval - group of 5. In *Proceedings of Coling-ACL*, pages 31–37, 1998.

[10] C. Morrell, J. Pearson, and L. Brant. Linear transformation of linear mixed effects models. *The American Statistician*, 51:338–343, 1997.

[11] 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.

[12] A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: an experiential study. *Journal of Program Languages*, 4(3):143–167, 1996.

[13] G. Verbeke and G. Molenberghs. *Linear mixed models for longitudinal data*. Springer-Verlag, New York, second edition, 2001.

[14] Webster. *Collegiate Dictionary,* 11^{th} *Edition*. Merriam-Webster, 2003.

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