TRIS: a Fast and Accurate Identifiers Splitting and Expansion Algorithm

Latifa Guerrouj[1], Philippe Galinier[1], Yann-Gaël Guéhéneuc[1], Giuliano Antoniol[1], Massimiliano Di Penta[2]

[1] DGIGL, École Polytechnique de Montréal
[2] Dept. of Engineering, University of Sannio, Italy

E-mail: {latifa.guerrouj, philippe.galinier, yann-gael.gueheneuc}@polymtl.ca, antoniol@ieee.org, dipenta@unisannio.it

***Abstract*—Understanding source code identifiers, by identifying words composing them, is a necessary step for many program comprehension, reverse engineering, or redocumentation tasks. To this aim, researchers have proposed several identifier splitting and expansion approaches such as Samurai, TIDIER and more recently GenTest. The ultimate goal of such approaches is to help disambiguating conceptual information encoded in compound (or abbreviated) identifiers.**

This paper presents TRIS, TRee-based Identifier Splitter, a two-phases approach to split and expand program identifiers. First, TRIS pre-compiles transformed dictionary words into a tree representation, associating a cost to each transformation. In a second phase, it maps the identifier splitting/expansion problem into a minimization problem, *i.e.*, the search of the shortest path (optimal split/expansion) in a weighted graph.

We apply TRIS to a sample of 974 identifiers extracted from JHotDraw, 3,085 from Lynx, and to a sample of 489 identifiers extracted from 340 C programs. Also, we compare TRIS with GenTest on a set of 2,663 mixed Java, C and C++ identifiers. We report evidence that TRIS split (and expansion) is more accurate than state-of-the-art approaches and that it is also efficient in terms of computation time.

***Keywords*-Identifier Splitting/Expansion, Program Comprehension, Linguistic Analysis, Optimal Path, Weighted Acyclic graph.**

I. Introduction

When software documentation is scarce, outdated, or simply unavailable, source code remains the only up-to-date source of information for software engineers. Identifiers (*e.g.*, names of classes, methods, parameters, or attributes, etc.) account for approximately more than half of linguistic information [1] contained in source code. Researchers agree on the identifiers role and relevance for maintenance and evolution tasks. Identifiers must therefore be carefully chosen, to reduce the time and effort needed by program comprehension, a step preliminary to any software re-engineering or evolution activity [2], [3], [4], [5].

Identifiers are often composed concatenating (possibly following underscore or Camel Casing rules) domain or application specific terms (*e.g.*, *browser* in a Web application), natural language words, often plain English words (*e.g.*, *user*), as well as jargon, abbreviated words and–or acronyms (*e.g.*, *execQuery*, *memPntr*, *SOA_Engine*). To understand the source code, a developers map identifiers onto high-level concepts they convey. Whenever an identifier is a compound one, a crucial and preliminary task developers have to mentally perform is that of splitting identifiers into component terms, possibly expanding abbreviations and acronyms composing them into words.

In the quest of supporting program understanding, source code redocumentation, and software reverse-engineering, the research community has developed various approaches to split identifiers into terms and expand terms into words. Beside the widely known and basic Camel Case split, it is worth to mention Samurai [6], TIDIER [7], and more recently Normalize [8] based on GenTest [9]. Different approaches have different strengths and weaknesses. On one hand, for Java source code, a simple Camel Case split algorithm and Samurai perform reasonably well [6]. On the other hand, neither Samurai nor Camel Case perform term expansion (*i.e.*, expand abbreviation such as *pntr* into the original word *pointer*). In addition, the accuracy of the latter techniques on C code is much worse than on Java, see for example [7]. TIDIER performs term expansion and the reported results [7] suggest that it outperforms Samurai. However, TIDIER has a cubic complexity in the number of characters composing the identifier, words in the dictionary, and maximum number of characters composing dictionary words. GenTest is known to be fast, however it generates all possible splittings for hard-words composing an identifier *i.e.*, it may generate an overwhelming number of possible splits.

This paper proposes TRIS (TRee-based Identifier Splitter), a novel two-phases approach to split and expand source code identifiers. TRIS takes as input a dictionary of words (*e.g.*, taken from an upper domain ontology), the source code of the program to analyze, containing identifiers to be split and expanded.

TRIS represents transformations as a rooted directory tree where every node is a letter and every path in the tree (from the root to a leaf) represent a transformation having a given cost. Based on such transformations, possible splittings and expansions of an identifier are represented as an acyclic direct graph, called an auxiliary graph where again nodes represent letters and edges represent transformation costs. Once such a graph is built, solving the optimal splitting and expansion problem means determining the shortest path—from the starting node to the ending node—on such a graph using a simplified Dijkstra's algorithm.

This paper describes the TRIS splitting and expansion approach, along with an empirical evaluation. To this aim, we evaluate TRIS on publicly available data sets used to evaluate previous approaches. We use a sample of 974 JHotDraw (Java) identifiers, 3,085 Lynx (C) identifiers [10]; a set of 489 C identifiers sampled from 340 GNU utilities [7]; and a set of 2,663 mixed Java, C, and C++ identifiers used in the study by Lawrie *et al.* [9]. Reported results show that TRIS performs more accurately than state of the art approaches and is efficient in terms of computation time.

This paper is organized as follows. Section II relates this work to the existing literature on the role of source code lexicon on software quality, and overviews existing identifier splitting/expansion approaches. Section III describes the TRIS approach. Section IV reports the empirical study to evaluate the proposed approach. Finally, Section V concludes the paper and outlines directions for future work.

II. Background and Related Work

Stemming from Deißenböck and Pizka's observation on the relevance of identifiers' terms for program comprehension [1], several approaches have been introduced to split and expand source code identifiers. In the following, we provide an overview of existing identifier splitting and expansion techniques.

A. Camel Case Splitting

Camel Case splitting is a simple and widely used pre-processing algorithm. It is based on the use of Camel Case and separators. This approach splits compound identifiers according to the following rules:

RuleA: Underscore, structure and pointer access, as well as special symbols are replaced with the space character.

RuleB: Identifiers are split where terms are separated using the Camel Case convention. For example, *userId* is split into *user* and *Id*, while *setGID* is split into *set* and *GID*.

RuleC: When two or more upper-case characters are followed by one or more-lower case characters, the identifier is split at the last-but-one upper-case character. For example, *SSLCertificate* is split into *SSL* and *Certificate*. Sometimes, a space is inserted before and after each sequence of digits. For example, *print_file2device* is split into *print*, *file*, 2, *device*.

Camel Case splitting cannot split same-case composite identifiers, such as *USERID* and *currentsize* into separate terms.

B. Samurai

Samurai [6] is a technique to split identifiers into their component terms by mining term frequencies in large source code bases. It relies on two assumptions:

1) A substring composing an identifier is also likely to be used in other parts of the program or in other programs alone or as a part of other identifiers.
2) Given two possible splits of a given identifier, the split that most likely represents the developer's intent partitions the identifier into terms occurring more often in the program.

Samurai uses its two frequency tables in conjunction with Camel Case rules. It first tries to apply Camel Case split and then ranks possible splits according to its identifiers frequency tables (one program specific and the other global). In this way, Samurai can deal with same-case identifiers, such as *USERID*, *currentsize*, or mixed-case ones (*e.g.*, *DEFMASKBit*).

C. TIDIER: Term IDentifier RecognizER

TIDIER [7] is an approach to split and expand program identifiers using high-level and domain concepts captured into multiple dictionaries. It is inspired by speech recognition techniques and uses a thesaurus of words and abbreviations, plus a string-edit distance between terms and words computed via Dynamic Time Warping. TIDIER's main assumption is that it is possible to mimic developers when creating an identifier relying on a set of words transformations. For example, to create an identifier for a variable that counts the number of software bugs, the two words, *number* and *bugs*, can be concatenated with or without an underscore, or following the Camel Case convention *e.g.*, *bugs_number*, *bugsnumber* or *bugsNumber*. Developers may drop vowels and (or) characters to shorten one or both words of the identifier, thus creating *bugsNbr* or *nbrOfbugs*. Similarly to Samurai, TIDIER exploits contextual information in the form of specialized dictionaries (*e.g.*, acronyms, contractions and domain specific terms) and mimics the process of transforming words via contraction rules.

D. GenTest and Normalize

GenTest [9] is a splitting algorithm that consists of two parts: generation and test. The generation part of GenTest generates all possible splittings; the test part, however, evaluates a scoring function against each proposed splitting. GenTest uses a set of metrics to characterize the quality of the split.

This work was refined to support the expansion of identifiers and led to Normalize [8]. Normalize deals with identifier expansion using a machine translation technique, the maximum coherence model. The heart of Normalize is a similarity metric computed from co-occurrence data. Co-occurrence data with context information is exploited to select the best candidate among several possible expansions.

We share with the above works the general problem of identifier split and expansion. Most recent identifier split and–or expansion approaches (*e.g.*, Samurai, TIDIER and GenTest) use in a way or the other contextual information, encoded as frequency tables, specific domain-dependent dictionaries, or as metrics to select the most likely split. TRIS is inspired by TIDIER, Samurai and GenTest. However, TRIS problem formulation and solution is novel and its implementation is fast and accurate.

III. The Approach

In this section, we first formalize the problem of finding the splitting-expansion of a given identifier as an optimization problem, then we describe in details the phases and steps implemented in TRIS.

Programmers often build source code identifiers by applying a set of transformation rules to words, such as dropping all vowels (*e.g.*, *pointer* becomes *pntr*), dropping one or more characters, or dropping a suffix (eg *allocation* becomes *alloc*) [7].

TRIS treats the identifier splitting and expansion as an optimization problem, in the following referred as Optimal Splitting-Expansion (OSE) problem. The search space of the OSE problem contains a set of solutions that represent potential splitting-expansions of the input identifier. Once a cost is assigned to each solution, the OSE problem consists in finding a solution with minimal cost, which, hopefully, corresponds to the correct splitting-expansion of the input identifier.

To efficiently resolve the OSE problem, TRIS applies a two-phase strategy. The first phase—named "building dictionary transformations"—builds a set of legal transformations based on an input dictionary using a family of transformation types. The obtained set of transformations is then compressed and represented as an arborescence *i.e.*, a directed rooted tree. The second phase is named "identifier processing". Its goal is to determine an optimal splitting-expansion of a given input identifier. Note that the second phase uses the directed rooted tree (*i.e.*, the arborescence) built during the first phase.

In practice, we generally want to find the splitting-expansion of a set of identifiers originating from the same source code—instead of a single one. It is very important to note that building the dictionary transformations (phase 1) has to be performed only once. Then, identifier processing (phase 2) will be applied to each identifier to be split/expanded.

As we will show in this section, the identifier processing algorithm boils down to finding a shortest path in an identifier graph. Its complexity is $\mathcal{O}(n^2)$, where n represents the size of the input identifier, this is to say quadratic in the length of the identifier to split. As a result, producing the splitting-expansion of a given input identifier is very fast—thousands of identifiers can be split within a few seconds. On the other hand, the creation of dictionary transformations (which is performed only once) can take a few seconds (*e.g.*, 35 milliseconds for a dictionary of 2,953 words on a machine having a processor running at 2.10 GHz and memory (RAM) of 6.00 GB).

A. TRIS Formalization of the Optimal Splitting-Expansion Problem

The input of the OSE problem consists of: (i) an identifier to be split/expanded; (ii) a dictionary (ii) source code of the system the identifier belongs to. In the following, we define the transformations, the search space, and the cost function of the OSE problem.

Transformations

We call *transformation* a couple ($wOrig \rightarrow\ w$) where $wOrig$ is a dictionary word and w a string. For example, the transformation ($pointer \rightarrow pntr$) indicates that the dictionary word *pointer* has been transformed into *pntr*.

The current implementation of TRIS uses four types of transformations:

1) *Null transformation:* keep the word as it is;
2) *Vowels removal:* remove all vowels contained in the dictionary word, *e.g.*, *pointer* $\rightarrow$ *pntr* but the first one if it is the first character of the identifier;
3) *Suffix removal:* suffixes such as *ing*, *tion*, *ed*, *ment*, and *able* are removed from the dictionary word, *e.g.*, *improvement* $\rightarrow$ *improve*;
4) *First* n *characters*: keeps only the first n characters of a word with n $\in \{3,4,5\}$, *e.g.*, *rectangle* $\rightarrow$ *rect* for $n = 4$.

In the following, we denote by type(.) the type of a given transformation.

Search Space

A potential solution (*i.e.*, an element of the search space) corresponds to a splitting-expansion. Such a solution is composed of a series of transformations. For example, a potential splitting-expansion of *drawimagrect* is (*draw*$\rightarrow$ $draw$)/(*image*$\rightarrow$ $imag$)/(*rectangle*$\rightarrow$ $rect$) as:

- the concatenation of *draw*, *imag*, and *rect* produces *draw.imag.rect*=*drawimagrect*;
- the words *draw*, *image*, and *rectangle* belong to a given dictionary;
- the transformations (*draw*$\rightarrow$ $draw$), (*image*$\rightarrow$ $imag$), and (*rectangle*$\rightarrow$ $rect$) are legal transformations (whose types are respectively 1, 4, and 4).

Cost Function

Recall that a solution (splitting-expansion) is composed of a series of transformations. The cost of a solution is simply the sum of the costs of the transformations it is composed of. Each transformation has an associated cost C($wOrig \rightarrow\ w$) defined as the sum of two terms:

$$\text{C}(wOrig \rightarrow\ w) = \alpha * \text{Freq}(wOrig) + \text{C}(\text{type}(wOrig \rightarrow\ w))$$

The first term Freq($wOrig$) is the relative frequency of the dictionary word $wOrig$ in the source code, multiplied by a parameter α. The frequency is simply the number of occurrences of a dictionary word in the source code, divided by the sum of all occurrences of dictionary words in source code. TRIS uses the relative frequency as a local context to determine the most likely identifier splitting-expansion.

The value of the parameter α is negative: as a result, a transformation ($wOrig \rightarrow\ w$) such that $wOrig$ has a low frequency will be in fact penalized.

The second component C(type($wOrig \rightarrow\ w$)) corresponds to the cost of the transformation type. The cost of the four different transformation types are algorithm parameters whose values will be reported in the empirical part of the paper; the general idea is to assign a low cost to a transformation type that is believed to be more natural and more often used by developers.

B. Building Dictionary Transformations Algorithm

The goal of this phase is to build the set of transformations and to represent it as an arborescence. It consists of the three following successive steps:
(1.1) Computation of the frequency of dictionary words;
(1.2) Construction of the set of transformations;
(1.3) Construction of the arborescence of transformations.
Each of these steps is detailed in the following. In this context, a dictionary is just a collection of strings representing application-level concepts (*e.g.*, socket), known acronyms (*e.g.*, cpu), and plain English words (*e.g.*, a set of WordNet entries).

Computation of the frequency of dictionary words (step 1.1)
Input: (1) a dictionary; (2) code of the application.
Output: frequencies of dictionary words.

During this step, the source code is scanned. For each string found in the source code, if this string corresponds to a dictionary word, we increment the number of occurrences of this word. Finally, this procedure returns the frequency of each dictionary word.

Construction of the set of transformations (step 1.2)
Input: (1) a dictionary; (2) frequencies of dictionary words.
Output: set of transformation triples.
For each dictionary word $wOrig$ and each type T of transformation (T=1..4), we determine each word w that can be derived from $wOrig$ according to T. For each transformed word w, we add the triplet ($wOrig$, w, C($wOrig$, w)) to the set of transformations.

Construction of the arborescence (step 1.3)
Input: set of transformation triples.
Output: arborescence of transformations.
The goal of this step is to represent the set of transformations (built during step 1.2) under the form of an arborescence. The rationale is that, in the following, it will dramatically decreases the complexity of the construction of the auxiliary graph (step 2.1).

In this arborescence, each node (except the root) is labeled with a letter of the alphabet. Each transformation triple ($wOrig$, w, $cost$) is represented by a path that starts from the root and whose nodes are labeled by the letters of w. The last node X contains a pointer towards the considered transformation triple. In fact, as several transformations ($wOrig_1$, w_1, $cost_1$), ($wOrig_2$, w_2, $cost_2$), etc. may produce the same string w, we only take into account one of those whose cost is minimum. An interesting property of this arborescence is that, given a string w, we can determine in $\mathcal{O}(|w|)$ if there exists at least one transformation ($wOrig$, w, $cost$) and, if it is the case, which is the transformation of minimal cost. Table I provides a simplified example of a (small) dictionary used

TABLE I
DICTIONARY TRANSFORMATIONS BUILDING INFORMATION FOR THE IDENTIFIER *callableint*.

Information for the Dictionary Transformations Building Phase		
Dictionary Words (D)	**Words Frequencies**	**Transformations Set**
d_1 = "$able$"	$f_1 = 0.1$	$t_1 = (d_1, abl, 0.55)$
		$t_2 = (d_1, able, -0.2)$
d_2 = "$call$"	$f_2 = 0.2$	$t_3 = (d_2, cal, 0.35)$
		$t_4 = (d_2, call, -0.4)$
		$t_5 = (d_2, cll, 0.6)$
d_3 = "$callable$"	$f_3 = 0.6$	$t_6 = (d_3, calla, -0.95)$
		$t_7 = (d_3, callable, -1.2)$
		$t_8 = (d_3, cllbl, -0.2)$
d_4 = "$interface$"	$f_4 = 0.1$	$t_9 = (d_4, int, 0.55)$
		$t_{10} = (d_4, inte, 0.3)$
		$t_{11} = (d_4, inter, -0.05)$
		$t_{12} = (d_4, interface, -0.2)$
		$t_{13} = (d_4, intrfc, 0.8)$

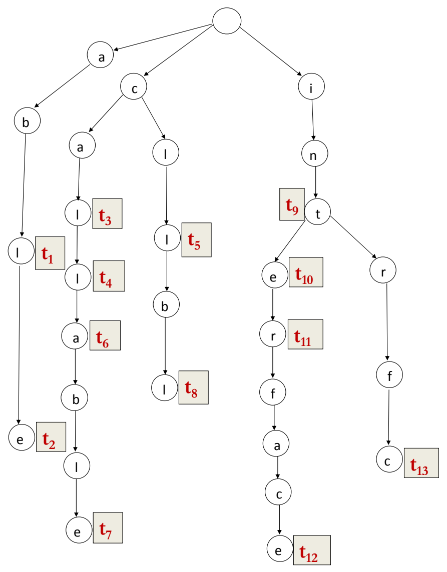

Fig. 1. Arborescence of Transformations for the Dictionary D.

to split/expand the identifier *callableint* along with dictionary words frequencies and the resulting transformation triples.

The arborescence of the transformations corresponding to the dictionary D used to split/expand the identifier *callableint* is shown in Fig. 1.

Let N be the sum of the sizes of w such that (*wOrig*, w, cost) belongs to the set of transformations. The arborescence construction algorithm complexity is $\mathcal{O}(N)$. Therefore, with respect to worst-case complexity, there is no extra-cost to transform the set of transformation triples into an arborescence.

C. Identifier Processing Algorithm

The goal of identifier processing is to determine an optimal splitting-expansion of a given input identifier *Idtf*.

It consists of the two following steps:
(2.1) Construction of the auxiliary graph associated to *Idtf*;
(2.2) Search for a shortest path in the auxiliary graph,

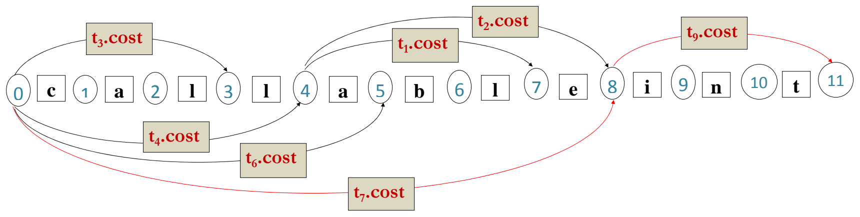

Fig. 2. Auxiliary Graph for the Identifier *callableint*.

corresponding to an optimal splitting-expansion of *Idtf*.

Construction of the auxiliary graph (step 2.1)

Input: (1) arborescence of transformations; (2) input identifier
Output: identifier auxiliary graph
Let *Idtf*`[i;j]` be the substring of *Idtf* between characters at position i and j. The auxiliary graph of *Idtf* is defined as follows:

- The graph has $|Idtf|$+1 vertices denoted by v_0,...,$v_{|Idtf|}$;
- For a transformation triple ($wOrig$, w, $cost$) such that w=Idtf`[i;j]`, there is an edge between the vertices v_i and v_j and the weight of this edge equals $cost$.

We can notice that a path in the auxiliary graph corresponds to a splitting-expansion of *Idtf*, and that the weight of this path corresponds to the cost of the corresponding splitting-expansion. Therefore, a shortest path in the graph corresponds to an optimal splitting-expansion.

More in detail, the auxiliary graph is built as follows. For every position p in the identifier *Idtf*, $p = 0...|Idtf|$, we go from the root of the arborescence of transformations and down following the path labeled by *Idtf*`[p;n]` where $n = |Idtf|$. For each node X on this path, if X.*transfPtr* is not null and points toward a transformation ($wOrig$, w, $cost$), we insert into the graph an edge between v_p and $v_{p+|w|}$ and assign to this edge a weight equals to *cost*. The complexity of this procedure is $\mathcal{O}(|n|^2)$, thus it is quadratic in the identifier length and as identifiers are usually short this step is very fast.

Fig. 2 shows the auxiliary graph corresponding to the identifier *callableint*, built using the arborescence shown in Fig. 1. On this example, we have two possible splits (based on the set of transformations shown in Table I). The first split is: *call-able-int*. The second is *callable-int*. Their corresponding expansions (pointing to their original dictionary words) are respectively *call-able-interface* (as int is derived from interface in the example), and *callable-interface*. According to the cost of transformations indicated in the last column of Table I, denoted (for simplification of Fig. 2) by t_i.cost with i $\in$ {1,...,13} (computed based on words frequencies shown in the second column of the same table, plus costs of used transformation types reported in section IV of the paper), the minimum cost is the one corresponding to the split *callable-int* and hence to the expansion *callable-interface*.

Search for an optimal splitting-expansion (step 2.2)

Input: (1) *Idtf* auxiliary graph
Output: an optimal splitting-expansion of *Idtf*
The auxiliary graph is acyclic. Therefore, although some edges may have negative weights (remember the α multiplier), it makes sense to talk about a shortest path in this graph. The shortest path found in the auxiliary graph provides us with an optimal splitting-expansion of *Idtf*. The complexity of this procedure is at worst $\mathcal{O}(|n|^2)$, where $n = |Idtf|$.

IV. Case Study

The *goal* of this study is to analyze the proposed identifier splitting and expansion approach *TRIS*, with the *purpose* of evaluating its ability to correctly split and expand compound identifiers. The *quality focus* is the accuracy of TRIS when splitting identifiers and expanding abbreviated terms (resulting from the splitting) with respect to oracles, and compared with other state-of-the-art approaches, namely a simple Camel Case Splitter, Samurai, TIDIER, and GenTest. The *perspective* is of researchers interested to develop an approach for identifier splitting and expansion, with the aim of easing program comprehension and maintenance tasks. The *context* consists of a set of identifiers extracted from Java, C and C++ programs. Specifically, we use (i) 974 identifiers extracted from the source code of JHotDraw, (ii) 3,085 identifiers from Lynx, (iii) 489 identifiers extracted from the source code of 340 C GNU Linux utilities, and (iii) a mixed set of Java, C, and C++ identifiers used in the study by Lawrie *et al.* [9] and made available on-line[1]. We used the latter data for replication purposes as we want to compare TRIS (in terms of splitting accuracy) with GenTest[2].

JHotDraw[3] is a Java framework for drawing 2D graphics. The project started in October 2000 with the main purpose of illustrating the use of design patterns in a real context.

Lynx[4] is a free, open-source, text-only Web browser and Gopher client. Lynx is entirely written in C. Its development began in 1992 and it is now available on several platforms, including Linux, Unix, and Windows. Table II reports the main characteristics of Lynx and JHotDraw analyzed releases.

The 340 C/C++ programs from which we sample 489

[1] *www.cs.loyola.edu/ binkley/ludiso*
[2] *http://splitit.cs.loyola.edu*
[3] *http://www.jhotdraw.org*
[4] *http://lynx.isc.org/*

TABLE II
MAIN CHARACTERISTICS OF JHOTDRAW AND LYNX.

JHotDraw and Lynx Systems		
	JHotDraw	Lynx
Analyzed Releases	5.1	2.8.5
Files	155	247
Size (KLOCs)	16	174
Identifiers (> 2 chars)	2,348	12,194

TABLE III
MAIN CHARACTERISTICS OF THE 340 PROJECTS FOR THE 489 RANDOMLY-SAMPLED IDENTIFIERS.

GNU Projects (337 Projects)				
	C	C++	.h	Java
Files	57,268	13,445	39,257	14,811
Size (KLOCs)	25,442	2,846	6,062	3,414
Terms	26,824	–	17,563	–
Identifiers	1,154,280	–	619,652	–
Oracle Identifiers	927	–	26	–
Linux Kernel				
	C	C++	.h	Java
Files	12,581	–	11,166	–
Size (KLOCs)	8,474	–	1,994	–
Terms	19,512	–	13,006	–
Identifiers	845,335	–	352,850	–
Oracle Identifiers	73	–	4	–
FreeBSD				
	C	C++	.h	Java
Files	13,726	128	7,846	15
Size (KLOCs)	1,800	128	8,016	4
Terms	21,357	–	12,496	–
Identifiers	634,902	–	278,659	–
Oracle Identifiers	20	–	0	–
Apache Web Server				
	C	C++	.h	Java
Files	559	–	254	–
Size (KLOCs)	293	–	44	–
Terms	6,446	–	3,550	–
Identifiers	33,062	–	11,549	–
Oracle Identifiers	11	–	0	–

identifiers are 337 GNU[5] projects, the Linux Kernel[6] 2.6.31.6, FreeBSD[7] 8.0.0, and the Apache Web server[8] 2.2.14. The main characteristics of these projects are listed in Table III.

The sample of data used by Lawrie *et al.* was randomly drawn from a source base that includes 186 programs, for a total of 26 MLOC of C, 15 MLOC of C++, and 7 MLOC of Java. Raw and oracle data sets are available on-line[9]. Details about the empirical evaluation can be found in a previous paper by Lawrie *et al.* [9].

As explained in Section III.A, the costs assigned to the introduced transformation types (second component of our cost function) are algorithm parameters. In our empirical study, we assigned 0 as a cost to the *null transformation*, (0.75, 0.5, and 0.25) as costs to the three transformations *keeping the n first characters* with n $\in$ {3,4,5} respectively. For the transformations *removing vowels and suffix removal*, we respectively assigned 1 and 1.5. Also, we assigned to the parameter α, -2 as a value. To determine the values of the parameters, we run TRIS multiple times with different transformations' costs and alpha values.

The study reported in this section aims at addressing the following research question:

> *What is the accuracy of the TRIS identifier splitting and expansion approach compared with alternative state-of-the art approaches?*

Specifically, we measure the TRIS performances by comparing the splitting/expansion of the automatic approaches with those of the manually built oracles, and by assessing the splitting/expansion accuracy in terms of correctness, precision, recall, and F-measure, as reported in Section IV-A. We then compare the performances of TRIS with those of Camel Case splitter, Samurai, TIDIER, and GenTest.

A. Variable Selection and Study Design

The main independent variable of our study is the splitting algorithm used. This factor has five possible levels: (1) TRIS (which is our experimental group), and four control groups, i.e., (2) Camel Case, (3) Samurai, (4) TIDIER, (5) GenTest.

The dependent variables considered in our study are *precision*, *recall*, and *F-measure*, and the *splitting correctness*. The splitting correctness tells whether an identifier was correctly split or not (with respect to the oracle). The correctness measure has the disadvantage of providing a Boolean evaluation of the splitting, *i.e.*, when the split is almost correct (most of the terms are correctly identified), then correctness would still be false. To overcome the limitation of the correctness measure and provide a more insightful evaluation of TRIS, we also report the precision, recall, and F-measure.

Given an identifier s_i to be split and expanded, $o_i = \{oracle_{i,1}, \ldots oracle_{i,m}\}$ the expansion in the manually-produced oracle, and $t_i = \{term_{i,1}, \ldots term_{i,n}\}$ the set of terms obtained by an approach, precision and recall are defined as follows:

$$precision_i = \frac{|t_i \cap o_i|}{|t_i|}, \quad recall_i = \frac{|t_i \cap o_i|}{|o_i|}$$

To provide an aggregated, overall measure of precision and recall, we use the F-measure, which is the harmonic mean of precision and recall:

$$\mathrm{F-measure} = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

In this paper, we generally measure and compare the performance of the various splitting algorithms in terms of precision, recall, and F-measure. The only case in which we use the correctness is when comparing with the oracle of GenTest, for which we only have correctness data available from [9].

B. Building the Oracles

As explained above, to evaluate the performances of TRIS (*i.e.*, precision, recall, and F-measure), we need an oracle, *i.e.*, for each identifier, a list of words obtained after splitting it and, wherever needed, after expanding abbreviated words. For example, a possible oracle for *drawRect* would be *draw rectangle*, obtained by splitting the identifier after the fourth

[5]http://www.gnu.org/
[6]http://www.kernel.org/
[7]http://www.freebsd.org/
[8]http://www.apache.org/
[9]*http://www.cs.loyola.edu/ binkley/ludiso/*

TABLE IV
DESCRIPTIVE STATISTICS OF THE USED PROGRAM-LEVEL DICTIONARIES FOR THE 340 GNU UTILITIES.

Dictionary level	Min	1Q	Median	3Q	Max	Avg	σ
Application	29	900	1,797	3,028	22,190	2,320	2,374

character and expanding the abbreviation *Rect* into *rectangle*. We build the oracles following a consensus approach *i.e.*, two authors independently proposed an identifier split and expansion, and disagreements were discussed when the outcome was different. We adapted this approach to minimize the bias and the risk of producing erroneous oracles.

To build our oracles, we selected 974 JHotDraw identifiers, 3,085 Lynx ones, and 489 C ones extracted from the above mentioned 340 C programs, these randomly-selected identifiers were composed ones for which it was possible to define a manual splitting/expansion. Our analysis does not consider identifiers that are single English word (therefore no splitting/expansion was needed), and cryptic ones *i.e.*, identifiers for which it was impossible to find a splitting/expansion. Examples of this category are a few identifiers extracted from Lynx source code, *e.g.*, *hmmm*, *ixoth*, or *gieszczykiewicz*. We manually split identifiers and mapped abbreviated splits terms into words, thus, creating oracles for JHotDraw, Lynx and the 489 C identifiers.

For the comparison between TRIS and GenTest we used an oracle provided by Lawrie *et al.* [9].

C. Analysis Method

Study research questions aim at understanding if TRIS helps in splitting and expanding identifiers and thus easing program comprehension. In fact, we assume that, given an identifier, there exists an exact subdivision of it into words that, possibly after being transformed and once concatenated, form the identifier.

To apply TRIS, we built an application-level dictionary for each program part of our study, i.e., for JHotDraw, Lynx, and for each one of the 340 programs from which we sampled the C identifiers. In addition, we enrich these dictionaries by the use of domain knowledge (*i.e.*, common abbreviations and acronyms, library functions, etc) as a previous work [7] showed that a dictionary containing (i) application-level terms, (ii) English dictionary words, and (iii) common abbreviations and acronyms, allows to obtain the best performances. Details about the construction of application-level dictionaries and the used domain knowledge can be found in [7]. More precisely we used: (i) a set of 105 acronyms used in computer science (*e.g.*, ansi, dom, inode, ssl, url), (ii) a set of 164 abbreviations collected among the authors used when programming in C (*e.g.*, bool for Boolean, buff for buffer, wrd for word), and (iii) a set of 492 C library functions (*e.g.*, malloc, printf, waitpid, access). The application-level dictionaries for JHotDraw and Lynx contain 2,289 and 2,953 dictionary words respectively, while descriptive statistics about the size of dictionaries for the 340 GNU utilities are reported in Table IV.

We filtered identifiers containing short (up to two letters) prefixes such as *f* in *fname* or *ly*. This is because such prefixes can lead to any dictionary word containing the character *f* or the string *ly*.

To compare TRIS with other algorithms (except GenTest) we use a non-parametric test for pair-wise median comparison, namely, the Wilcoxon paired test. We use a paired test as our samples are dependent, as we compute, for each identifier, the difference of precision, recall and F-measure between the different approaches. The Wilcoxon test indicates whether the median difference between two approaches is significantly different from zero *i.e.*, $H_0 : \mu_d = 0$, where μ_d is the median of the differences.

Finally, in addition to the statistical comparison, we compute the effect-size of the difference using Cliff's delta non-parametric effect size measure [11], defined as the probability that a randomly-selected member of one sample has a higher response than a randomly selected member of a second sample, minus the reverse probability. Cliff's delta ranges in the interval $[-1, 1]$ and is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$.

Since we execute the Wilcoxon paired test multiple times to compare the various approaches, we must correct significant p-values. We use the Holm correction [12], which is similar to the Bonferroni correction, but less stringent. It works as follows: (i) the p-values obtained from multiple tests are ranked from the smallest to the largest, (ii) the first p-value is multiplied by the number of tests performed (n), and is deemed to be significant if it is less than 0.05, and (iii) the second p-value is multiplied by $n - 1$, and so on.

For what concerns the comparison with GenTest, since the comparison is performed in terms of correctness (which is a categorical variable), we use Fisher's exact test [13] which compares proportion of correct and non correct splittings provided by TRIS and GenTest. To quantify the effect size of the difference between the two approaches, we also computed the *odds ratio* (OR) [13] indicating the likelihood of an event to occur, defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the percentage of identifiers correctly split by TRIS (experimental group), to the odds q of it occurring in the other sample, *i.e.*, the percentage of identifiers correctly split by GenTest (control group): OR $= \frac{p/(1-p)}{q/(1-q)}$. An OR of 1 indicates that the event is equally likely in both samples. OR > 1 indicates that the event is more likely in the first sample (TRIS) while an OR < 1 indicates the opposite (GenTest).

D. Study Results

This section reports the results of the empirical study. Table V reports descriptive statistics (1st quartile, median, 3rd quartile, mean, standard deviation) of the accuracy of TRIS and the ones of Camel Case, Samurai, and TIDIER. Results of the statistical tests for JHotDraw are reported in Table VI. Similarly, descriptive statistics and statistical test results for Lynx are reported in Tables VII and VIII respectively.

TABLE V
PRECISION, RECALL AND F-MEASURE OF TRIS, CAMEL CASE, SAMURAI, AND TIDIER ON JHOTDRAW.

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	Camel Case	1.0000	1.0000	0.9244	1.0000	0.2424
	Samurai	1.0000	1.0000	0.9316	1.0000	0.2244
	TIDIER	1.0000	1.0000	0.9716	1.0000	0.1472
	TRIS	1.0000	1.0000	0.9804	1.0000	0.2025
Recall	Camel Case	1.0000	1.0000	0.9203	1.0000	0.2502
	Samurai	1.0000	1.0000	0.9367	1.0000	0.2129
	TIDIER	1.0000	1.0000	0.8984	1.0000	0.2158
	TRIS	1.0000	1.0000	0.9084	1.0000	0.1213
F-measure	Camel Case	1.0000	1.0000	0.9217	1.0000	0.2476
	Samurai	1.0000	1.0000	0.9325	1.0000	0.2200
	TIDIER	1.0000	1.0000	0.9233	1.0000	0.1791
	TRIS	1.0000	1.0000	0.9328	1.0000	0.1614

TABLE VI
COMPARISON AMONG APPROACHES: RESULTS OF WILCOXON PAIRED TEST AND CLIFF'S DELTA EFFECT SIZE ON JHOTDRAW.

Approach 1	Approach 2	adj p-value	Cliff's delta
TRIS	Camel Case	0.431	0.041
TRIS	Samurai	0.894	0.001
TRIS	TIDIER	**0.024**	0.043

TABLE VII
PRECISION, RECALL AND F-MEASURE OF TRIS, CAMEL CASE, SAMURAI, AND TIDIER ON LYNX.

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	Camel Case	0.0000	0.5000	0.4065	0.7500	0.4147
	Samurai	0.0000	0.5000	0.4767	1.0000	0.4089
	TIDIER	0.8000	1.0000	0.8609	1.0000	0.2674
	TRIS	1.0000	1.0000	0.9344	1.0000	0.1369
Recall	Camel Case	0.0000	0.3333	0.3705	0.6667	0.4066
	Samurai	0.0000	0.3333	0.4569	1.0000	0.4101
	TIDIER	0.7500	1.0000	0.8499	1.0000	0.2684
	TRIS	1.0000	1.0000	0.9138	1.0000	0.2060
F-measure	Camel Case	0.0000	0.4000	0.3851	0.7273	0.4086
	Samurai	0.0000	0.4000	0.4634	1.0000	0.4084
	TIDIER	0.6667	1.0000	0.8525	1.0000	0.2664
	TRIS	1.0000	1.0000	0.9206	1.0000	0.2055

TABLE VIII
COMPARISON AMONG APPROACHES: RESULTS OF WILCOXON PAIRED TEST AND CLIFF'S DELTA EFFECT SIZE ON LYNX.

Approach 1	Approach 2	adj p-value	Cliff's delta
TRIS	Camel Case	**<0.001**	0.743
TRIS	Samurai	**<0.001**	0.684
TRIS	TIDIER	**<0.001**	0.204

Results indicate that, for JHotDraw, TRIS achieves $F-measure = 0.9328$, Camel Case $F-measure = 0.9217$, Samurai $F-measure = 0.9325$, and TIDIER $F-measure = 0.9233$. Not surprisingly, Camel Case and Samurai work well enough on JHotDraw, because JHotDraw developers carefully adhered to coding standards and identifier creation rules. Also, TIDIER performs almost similarly to them, even if its approach does not necessarily reward the use of coding standards as for instance Camel Case does. Statistical comparisons of Table VI show that (i) there is no significant difference between TRIS, Camel Case, and Samurai on JHotDraw; and (ii) TRIS performs significantly better than TIDIER with a very small effect size, $d < 0.148$. On Lynx, in terms of F-measure, TRIS significantly outperforms ($F-measure = 0.9206$) Camel Case ($F-measure = 0.3851$), Samurai ($F-measure = 0.4634$), and TIDIER ($F-measure = 0.8525$). More precisely, the statistical comparisons of Table VIII indicate that, on Lynx (i) TRIS significantly outperforms the Camel Case splitter ($d = 0.743$) and Samurai ($d = 0.684$), and (ii) TRIS performs significantly better than TIDIER with a small effect size ($d = 0.204$).

Table IX reports the performances of TRIS on the sample of 489 C identifiers. It also reports TIDIER accuracy on this set. On such data set, we do not report performances of Camel Case and Samurai, since it is known from [7] that TIDIER outperforms Camel Case and Samurai on C systems when using application-level dictionaries augmented with domain knowledge. Hence, we are only interested to compare TRIS with the approach performing better on this data set *i.e.*, TIDIER. Results show that, in terms of F-measure, TRIS performs better ($F-measure = 0.879$) than TIDIER ($F-measure = 0.6409$) for this set also. The statistical comparison through Wilcoxon test indicates that the difference is statistically significant (p-value < 0.001), and that the Cliff's delta effect size is medium ($d = 0.456$).

Table X reports the results of TRIS, in terms of precision, recall, and F-measure, on the data set from Lawrie *et al.* [9]. As it can be noticed, performances are very high, with a median of 100% and a mean precision of 98%, recall of 94% and F-measure of 96%.

Table XI reports the accuracy of TRIS in terms of percentage of correct splittings, compared with the performances of GenTest and Samurai as reported by Lawrie *et al.* [9]. As it can be noticed, TRIS correctly splits identifiers in 86% of the cases, while GenTest does it in 82% of the cases, and Samurai in 70% of the cases.

When comparing the correctness of TRIS with the one of GenTest, Fisher's exact test does not indicate a significant difference (p-value=0.5), even though the achieved correctness is higher for TRIS. We believe the comparison would be insightful if precision, recall or F-measure were provided because splitting correctness, in our case, is a Boolean variable that returns (true) if the split is correct and (false) if not. Thus, when the splitting is almost correct, *i.e.*, most of the terms are correctly identified, the correctness would still be false. Unfortunately, this was the case for several identifiers in the study of Lawrie *et al.*. Examples of such identifiers are the ones prefixed with letters (*e.g.*, *mEnvironmental-*

TABLE IX
PRECISION, RECALL AND F-MEASURE OF TRIS AND TIDIER ON THE 489 C SAMPLED IDENTIFIERS.

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	TIDIER	0.4000	0.6667	0.6368	1.000	0.3681
	TRIS	1.0000	1.0000	0.8933	1.0000	0.2471
Recall	TIDIER	0.5000	0.6667	0.6496	1.000	0.3654
	TRIS	1.0000	1.0000	0.872	1.0000	0.2606
F-measure	TIDIER	0.4000	0.6667	0.6409	1.0000	0.3650
	TRIS	1.0000	1.0000	0.879	1.0000	0.2524

TABLE X
PRECISION, RECALL AND F-MEASURE OF TRIS ON THE DATA SET FROM LAWRIE *et al.*.

Metric	Approach	1Q	Median	Mean	3Q	σ
Precision	TRIS	1.0000	1.0000	0.9763	1.0000	0.1184
Recall	TRIS	1.0000	1.0000	0.9439	1.0000	0.1565
F-measure	TRIS	1.0000	1.0000	0.9559	1.0000	0.1358

TABLE XI
CORRECTNESS OF THE SPLITTING PROVIDED USING THE DATA SET FROM LAWRIE *et al.*.

Approach	Identifier Splitting Correctness
Samurai	70%
GenTest	82%
TRIS	86%

istNb, *sOS_DriveDirectory*, *xGetJobStaus*, *xgetAutomaticFocus*, *xgetColumnWidth*, etc.) and that we filtered as the letters can be generated by any dictionary word prefixed with them. Also, even though the difference in the strict correctness measure is not high (86%) against (82%) for GenTest, the F-measure of our approach attains 96%. The latter measure clearly shows that the novel approach performs well on the overall data of Lawrie *et al.*. Also, while GenTest generates all the possible splittings for each hard-word of an identifier, the suggested approach only returns the split with the minimum cost using a tree-based representation that makes it efficient in terms of computation time.

E. Discussion

Quantitative results reveal the importance of TRIS for languages that are not promoted by the use of a consistent naming convention (*e.g.*, Java coding standards such as Camel Case or underscore). Example of such languages are C and C++ where developers tend to use many word abbreviations. These results confirm the ones reported in [7] and [10], in which the authors showed that Camel Case, Samurai and TIDIER perform in a similar way on Java but not C. In contrast to Java code where coding standards are often strictly followed, C coding standards such as the Indian Hill[10] coding standards or the GNU coding standards[11] do not enforce Camel casing; that is why both the novel approach and TIDIER outperforms Camel Case and Samurai (approaches that strictly follow Java coding standards) on Lynx. Furthermore, the novel approach is more accurate than TIDIER.

Wrong splittings provided by TRIS are due to identifiers containing acronyms or short abbreviations. For example, we believe that it is impossible to correctly split and expand acronyms such as *afaik* or *imho*. We also believe that even if we consider the context (*i.e.*, the frequency of dictionary words in the source code) in TRIS, it is impossible to find the exact expansion of identifiers prefixed with letters such as *f* in the identifier *fsize* because the mapping could vary from *file size* to *figure size* depending on the JHotDraw code region where *fsize* appears. Overall, the results show that the novel approach performs accurately than previous ones on the overall studied systems.

In addition to splitting and expansion performances, TRIS has the advantage of performing reasonably fast: it takes 0.049 seconds to compile the JHotDraw dictionary (of 2,289 words) and 3.709 seconds to split/expand the 974 JHotDraw identifiers, while it takes 0.053 seconds to compile the Lynx dictionary (of 2,953 words) and 16.940 seconds to process the 3085 Lynx identifiers. In fact, TIDIER computation time increases with the increase of the dictionary size due to its cubic distance evaluation cost plus the search time. Camel Case splitter and Samurai performs fast their computations. Yet, they are not accurate when naming conventions are not used. Also, with the above timing performance, TRIS is able to ensure it has a higher correctness than GenTest.

In summary, we can conclude the study stating that:

For Java programs properly following coding standards, a simple Camel Case splitter is enough. For C programs, TRIS outperforms Camel Case splitter, Samurai and TIDIER. Also, TRIS is slightly better than GenTest in terms of effect size; it has 1.34 times the chances of better splitting identifiers than GenTest, although the difference is not statistically significant.

F. Threats to Validity

In this section, we present several threats to validity associated with our evaluation.

Threats to *construct validity* concern the relation between the theory and the observation. In this paper, this is mainly due to possible mistakes in the oracles. We cannot guarantee that some identifiers could have been split and expanded in different ways by developers that originally created them, as we might have a limited knowledge of the developers intent. To limit this threat, we used different sources of information such as comments, source code context, and online documentation when producing the oracles. Another threat could be the fact that a given string can be derived from several dictionary words, *e.g.*, the string *imag* can be derived from *image* and *imagination* by applying word transformations. We mitigated such a threat by considering the identifier context, that is the frequency of source code strings.

Threats to *internal validity* concern any confounding factors that could have influenced our study results. Subjectivity in building the oracles are limited by letting two independent persons generating such oracles and comparing the results. The choice of dictionaries could also influence results. Nevertheless, we used a combination (application-level dictionary + English words + a set of domain-specific acronyms and abbreviations), which proven to produce the best performance in a previous work [7].

Threats to *conclusion validity* concern the relations between the treatment and the outcome. Proper tests were performed to address our research questions. In particular, we used non-parametric tests (Wilcoxon and Fisher's exact tests) that do

[10]*http://www.chris-lott.org/resources/cstyle/*
[11]*http://www.gnu.org/prep/standards/*

not make any assumption on the underlying distributions of the data. Also, we based our conclusions not only on the presence of significant differences but also on the presence of a practically relevant difference, estimated by means of an effect-size measure. Finally, we dealt with problems related to performing multiple Wilcoxon tests using the Holm's correction procedure. Last, but not least, other than showing statistical significance, we provide quantitative information about the practical relevance of the observed differences in terms of effect size (Cliff's delta or OR, depending on the test performed).

Threats to *external validity* concern the possibility of generalizing our results. To make our results as generalizable as possible, we analyzed Java, C, and C++ identifiers. We believe the number of the analyzed systems is sufficient enough to generalize our results. However, we cannot be sure that our findings will be valid for other domains, applications, or programming languages.

V. Conclusion

Often, identifiers are made up by concatenating English terms and–or acronyms and abbreviations. Identifying terms composing identifiers and mapping them to their corresponding domain concepts is a non-trivial task especially when naming conventions are not followed or when abbreviations are introduced by developers on an ad-hoc basis.

In this paper, we presented a two-phases approach named TRIS to split and expand identifiers. TRIS takes as input a dictionary of words, the identifiers to split, and the application source code. First, TRIS pre-compiles transformed dictionary words into a tree representation associating to each transformation a cost. Then, in a second phase, it maps the identifier splitting and expansion problem in a graph optimization (minimization) problem to find the optimal path (*i.e.*, the optimal splitting-expansion) in an acyclic weighted graph representing the identifier and all possible transformations contributing to the identifier sub-strings.

As reported in the paper, we apply TRIS on a sample of 974 identifiers extracted from JHotDraw (Java), 3,085 Lynx (C) identifiers, and on a sample of 489 C identifiers extracted from 340 C programs, and compare its results with those of a simple Camel Case splitter, and with alternative approaches, *i.e.*, Samurai [6] and TIDIER [7]. Then, we compare TRIS with GenTest on a set of 2,663 mixed Java, C, and C++ identifiers used by Lawrie *et al.* to evaluate GenTest [9] accuracy.

Results indicate that while for Java systems following appropriate naming conventions—such as JHotDraw—simple splitting approaches such as Camel Case are just enough, on C systems, TRIS significantly outperforms Camel Case, Samurai, and TIDIER with a medium to large effect size. In addition, TRIS performs better than GenTest in terms of accuracy on a data set from Lawrie *et al.* consisting in Java, C, and C++ identifiers, even though the difference in accuracy is small (4%). Oppositely to GenTest, which generates all possible splittings, TRIS uses a tree-based representation that makes it—in addition to being more accurate than other approaches—efficient in terms of computation time. In fact, TRIS produces one optimal split and expansion fast using an identifier processing algorithm having a quadratic complexity in the length of the identifier to split/expand.

Future work will extend TRIS evaluation, study the impact of different transformation types as well as the associated transformation costs. Finally, we would like to investigate the impact of software evolution on words frequencies drift, and thus the impact of software evolution on TRIS when frequencies drift but the dictionary is not recompiled. Last but not least, we would investigate other types of transformations to further improve TRIS accuracy.

References

[1] F. Deissenbock and M. Pizka, "Concise and consistent naming," in *Proc. of the International Workshop on Program Comprehension (IWPC)*, May 2005.

[2] A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experiential study," *Journal of Program Languages*, vol. 4, no. 3, pp. 143–167, 1996.

[3] B. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *Proc. of the Working Conference on Reverse Engineering (WCRE)*, Atlanta Georgia USA, October 1999, pp. 112–122.

[4] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.

[5] ——, "What's in a name? a study of identifiers," in *Proceedings of 14th IEEE International Conference on Program Comprehension*. Athens, Greece: IEEE CS Press, 2006, pp. 3–12.

[6] E. Enslen, E. Hill, L. L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009, Vancouver, BC, Canada, May 16-17, 2009*, 2009, pp. 71–80.

[7] L. Guerrouj, M. Di Penta, G. Antoniol, and Y. G. Guéhéneuc, "TIDIER: An identifier splitting approach using speech recognition techniques," *Journal of Software Maintenance - Research and Practice*, p. 31, 2011.

[8] D. Lawrie and D. Binkley, "Expanding identifiers to normalize source code vocabulary," in *Proc. of the International Conference on Software Maintenance (ICSM)*, 2011, pp. 113–122.

[9] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *Proc. of the Working Conference on Reverse Engineering (WCRE)*, 2010, pp. 112–122.

[10] N. Madani, L. Guerrouj, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "Recognizing words from source code identifiers using speech recognition techniques," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010), March 15-18 2010, Madrid, Spain*. IEEE CS Press, 2010.

[11] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[12] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal of Statistics*, vol. 6, pp. 65–70, 1979.

[13] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.