Normalizing Source Code Vocabulary to Support Program Comprehension and Software Quality

Latifa Guerrouj
DGIGL - SOCCER Lab, Ptidej Team
Polytechnique Montréal, Québec, Canada
latifa.guerrouj@polymtl.ca

***Abstract*—The literature reports that source code lexicon plays a paramount role in program comprehension, especially when software documentation is scarce, outdated or simply not available. In source code, a significant proportion of vocabulary can be either acronyms and-or abbreviations or concatenation of terms that can not be identified using consistent mechanisms such as naming conventions.**

It is, therefore, essential to disambiguate concepts conveyed by identifiers to support program comprehension and reap the full benefit of Information Retrieval-based techniques (*e.g.*, feature location and traceability) whose linguistic information (*i.e.*, source code identifiers and comments) used across all software artifacts (*e.g.*, requirements, design, change requests, tests, and source code) must be consistent.

To this aim, we propose source code vocabulary normalization approaches that exploit contextual information to align the vocabulary found in the source code with that found in other software artifacts. We were inspired in the choice of context levels by prior works and by our findings. Normalization consists of two tasks: splitting and expansion of source code identifiers. We also investigate the effect of source code vocabulary normalization approaches on software maintenance tasks.

Results of our evaluation show that our contextual-aware techniques are accurate and efficient in terms of computation time than state of the art alternatives. In addition, our findings reveal that feature location techniques can benefit from vocabulary normalization when no dynamic information is available.

***Index Terms*—Source code linguistic analysis, information retrieval, program comprehension, software quality**

I. Research Context: Program Comprehension and Software Quality

Understanding source code is a necessary step for many program comprehension, reverse engineering, or re-documentation tasks. In source code, identifiers (e.g., names of classes, methods, parameters, or attributes, etc.) account for approximately more than half of linguistic information [1].

Many researchers have shown the usefulness of source identifiers to enhance program comprehension and software quality and their relevance for maintenance and evolution [2], [3], [4], [5]. Other research works have used identifiers to recover traceability links [6], [7], [8], measure conceptual cohesion and coupling [9], [10], and, in general, as an asset that can highly affect source code maintainability [2], [4], [5].

Other researchers [11], [12] assessed the quality of identifiers, and comments, plus the information carried by the terms that compose them. They all concluded that identifiers, if carefully chosen, reflect the semantics and the role of the named entities they are intended to label.

Stemming from Deißenböck and Pizka observation on the significance of identifiers to capture programmers' intent and encode knowledge in software. Many works have been achieved to identify concepts embedded in identifers. To the best of our knowledge, these families are: CamelCase, the de-facto splitting algorithm based on the use of naming conventions. Samurai [13], an approach that relies on a lexicon and uses greedy algorithms to identify component words, plus GenTest [14] and Normalize [15]. GenTest is a splitting algorithm that generates all possible splittings and evaluates a scoring function against each proposed splitting; it uses a set of metrics to characterize the quality of the split. Normalize is a refinement of GenTest towards the expansion of identifiers. 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 expansion.

The novel approaches we suggested to tackle our challenge are TIDIER and TRIS. TIDIER is inspired by speech recognition technique and exploits context in the form of specialized dictionaries [16]. TRIS is inspired by TIDIER, however, its problem formulation and solution is novel and its implementation is fast and accurate. It models the problem as an optimization (minimization) one *i.e.*, the search of the shortest path (optimal split/expansion) in a weighted graph [17]. We also studied the effect of source code vocabulary normalization on feature location techniques. Results of our study outline potential benefits of putting additional research efforts into defining more sophisticated source code preprocessing techniques as they are useful in situations where execution information cannot be easily collected [18]. Currently, we are targeting other applications, namely, traceability and labeling of execution traces segments. Also, we are preparing additional studies to investigate the effect of source code lexicon on program comprehension and software quality.

II. Overarching Question of our Thesis

The research question of our thesis can be stated as follows:

How to automatically disambiguate concepts conveyed by identifiers using context to help developers understand programs and support software maintenance tasks, and to what extent our techniques will enhance program comprehension and software quality?

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

III. ANSWER

To answer our main research question, we first tackled the source code vocabulary normalization approaches part. Our initial findings is TIDIER [16], a technique that uses a thesaurus of words and abbreviations, plus a string-edit distance between terms and words computed via Dynamic Time Warping. TIDIER 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. It also assumes that developers may drop vowels and (or) characters to shorten one or both words of the identifier, thus creating bugsNbr or nbrOfbugs. TIDIER works well in comparison with approaches that have been suggested prior to it. It attained its best performances when using contextual-aware dictionaries enriched with domain knowledge. However, TIDIER computation time increases with the increase of the dictionary size due to its cubic distance evaluation cost plus the search time. That is why we suggested a novel approach, TRIS. TRIS takes as input a dictionary of words (*e.g.*, taken from an upper domain ontology), and the source code of the program to analyze. It represents transformations as a rooted directory tree where every node is a letter and every path in the tree 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 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 in the identifier graph.

To analyze the impact of sophisticated source code vocabulary normalization approaches on software maintenance, we first applied a set of techniques CamelCase, Samurai [13] and TIDIER [16] on two feature location techniques, one based on Information Retrieval and the other one based on the combination of Information Retrieval and dynamic analysis, for locating bugs and features. The results show that feature location techniques using Information Retrieval can benefit from better preprocessing algorithms in some cases, and that their improvement in effectiveness while using manual splitting over state-of-the-art approaches is statistically significant in those cases. However, the results for feature location technique using the combination of Information Retrieval and dynamic analysis do not show any improvement while using manual splitting, indicating that any preprocessing technique will suffice if execution data is available [18].

IV. METHODOLOGY

To answer our research question, we follow a methodology where the main phases are detailed below. Details about our achieved works are found in the corresponding publications.

1. Building Dictionaries: To map terms or transformed words composing identifiers to dictionary words, we build contextual-aware dictionaries containing words and terms belonging to the applications, known acronyms/abbreviations, and library functions [16].

2. Building Oracles: To validate our approach, we need an oracle. This means that for each identifier, we will have a list of terms obtained after splitting it and, wherever needed, expanding contracted words. The oracle is produced following a consensus approach: (i) a splitting of each sampled identifier, and expanded abbreviations is produced independently (ii) In a few cases, disagreements were discussed among all the authors. We adapted this approach in order to minimize the bias and the risk of producing erroneous results. This decision was motivated by the complexity of identifiers, which capture developers domain and solution knowledge, experience, and personal preference [16], [18].

3. Empirical Evaluation: To evaluate our suggested approaches, we conducted empirical studies where the quality focus is the precision, recall, and F-Measure. We applied TIDIER on identifiers sampled from benchmark of 340 C identifiers [16]. We used the same benchmark to evaluate TRIS, plus two other systems JHotDraw, Lynx, and a set of Java, C, and C++ identifiers used by Lawrie et al. [14]. To statistically compare our approaches with state of the art ones, we performed statistical tests, namely, Wilcoxon and Fisher exact tests. W also provide quantitative information about the practical relevance of the observed differences in terms of effect size (Cliffs delta or Odd-Ratio, depending on the test performed).

To analyze if sophisticated splitting and expansion would still be helpful to improve accuracy of feature location techniques applied in different scenarios and settings, we performed an empirical evaluation on two feature location techniques: one based on Information Retrieval and the other one based on the combination of Information Retrieval and dynamic analysis, for locating bugs and features of two open-source systems, Rhino and jEdit. Results of our extensive empirical evaluation can be found in [18]. We are currently following the same methodology for traceability. We wish to run more empirical studies on open-source projects, try to analyze patches and bug reports to see wether there are some bugs introduced due to the use of identifiers. This points will helps us bring some empirical evidence on the intuition that source code lexicon impacts software quality.

4. Experimental Studies: To prove the relevance of context for source code vocabulary normalization tasks, we conducted an experiment with 42 subjects, including bachelor, master, Ph.D. students, and post-docs. We randomly sampled a set of 50 identifiers from a corpus of open source C programs and asked subjects to split and expand 40 different identifiers using internal and external contexts: (i) source code functions, (ii) source code files, and (iii) source code files, plus a thesaurus of acronyms and abbreviations.

V. On-Going Work and Forcast Completion

At this time of our thesis, we achieved the main phases of our project: we developed accurate approaches to tackle our problem, we run experimental studies with developers to know what components are of interest for our approaches, and finally we started applying our techniques. We applied one of them with previous ones on feature location and we are currently investigating them on traceability [6]. Also, we are studying the evolution and quality of source code lexicon.

Our on-going work consists on addressing the research questions that have not been addressed yet, running more empirical studies on other applications and systems. Also, we would like to run more studies on how developers formulate and use identifiers to further generalize the results of TIDIER since word transformations may not be helpful for other software such as mathematical one where a quite number of variables as i, j, and k are declared. In addition, we plan to perform more experiments with subjects on identifiers and their impact of different program comprehension and software evolution tasks. Our final goal would be to integrate our techniques in an Integrated Development Environment to help developers write consistent identifiers and hence improve quality of their code.

VI. Related Work

In this section, we describe the most relevant contributions to our research project.

Early work [19], [20] on program comprehension and mental models—programmers' mental representation of the program being maintained—highlighted the significance of textual information to capture and encode programmers' intent and knowledge in software. The role of identifier naming was also investigated by Anquetil *et al.* [21], who suggested the existence—in the source code lexicon—of "hard-terms" that encode core concepts.

Takang *et al.* [2] empirically studied the role of identifiers and comments on source code understanding. They compared abbreviated identifiers to full-word identifiers and uncommented code to commented code. The results of their study showed that commented programs are more understandable than non-commented programs and that programs containing full-word identifiers are more understandable than those with abbreviated identifiers.

Caprile and Tonella [3] performed an in-depth analysis of the internal structure of identifiers. They showed that identifiers are an important source of information about system concepts and that the information they convey is often the starting point of program comprehension. Other researchers [11], [12] assessed the quality of identifiers, their syntactic structure, plus the information carried by the terms that compose them.

Deißenböck *et al.* [1] provided a set of guidelines to produce high-quality identifiers. With such guidelines, identifiers should contain enough information for a software engineer to understand the program concepts.

Lawrie *et al.* [4] attempted to assess the quality of source code identifiers. They suggested an approach, named QALP (Quality Assessment using Language Processing), relying on the textual similarity between related software artifacts. The QALP tool leverages identifiers and related comments to characterize the quality of a program. The results of their empirical study indicated that full words as well as recognizable abbreviations contribute to better program understanding. Their work suggested that the recognition of words composing identifiers, and, thus, of the domain concepts associated with them could contribute to a better comprehension.

Binkley *et al.* [22] investigated the use of the identifier separators, namely the Camel Case convention and underscores in program comprehension. They found that the Camel Case convention led to better understanding than underscores, and when subjects are properly trained, that subjects performed faster with identifiers built using the Camel Case convention rather than those with underscores.

Overall, prior work reveals that identifiers represent an important source of domain information, and that meaningful identifiers improves software quality and reduce the time and effort to acquire a basic comprehension level for any maintenance task.

Many cognitive models have been proposed in the literature and they all rely on the programmers' own knowledge, the source code and available documentation [20]. Disparities between various comprehension models can be explained in terms of differences in experimental factors such as programmer characteristics, program characteristics and task characteristics that influence the comprehension process [23].

Robillard *et al.* [24] performed an exploratory study to assess how developers investigate context, more precisely, source code when performing a software change task. Their study involved five developers performing a change task on a medium-size open source system. The main outcome of their study supports the intuitive notion that a methodical and structured approach to program investigation is the most effective. Their main findings related to our work is the fact that prior to performing a task, developers must discover and understand the subset of the system relevant to the task. Thus, task context is important to understand the task at hand and avoid information overload.

Kersten *et al.* [25] presented a mechanism that captures, models, and persists the elements and relations relevant to a task. They showed how their task context model reduces information overload and focuses a programmers' work by filtering and ranking the information presented by the development environment. They implemented their task context model as a tool, Maylar, for the Eclipse development environment. In their paper, a task context represents the program elements and relationships relevant to completing a particular task.

Sillito *et al.* [26] provided an empirical foundation for tool design based on an exploration of what programmers need to understand and of how they use tools to discover that information while performing a change task. The results of their study point to the need to move tools closer to

programmers' questions and the process of answering those questions and also suggest ways that tools can do this, for example, by maintaining and using more types of context and by providing support for working with larger and more diverse groups of entities and relationships.

We share with the above mentioned works the idea that source code lexicon plays a paramount role in program comprehension and software quality. We also agree that task context is important when performing source code vocabulary normalization approaches. Our goal, however, is to show the extent to which contextual source code vocabulary normalization techniques can improve program comprehension and software quality.

VII. Conclusion

Our initial contributions yields to accurate approaches for source code vocabulary normalization, namely, TRIS. TRIS formalizes the studied problem as an optimization problem where the aim is to find optimal the splitting/expansion in acyclic graph. TRIS exploits context because it has been proven by prior works and also by our findings that contextual information is relevant for our problem. Application of sophisticated normalization approaches on feature location techniques show that such tasks could be still helpful when no dynamic information is available. We are currently evaluating our techniques on additional software maintenance tasks. Our future work will focus on indepth studies for enhancing program comprehension and software quality using source code vocabulary normalization.

VIII. Acknowledgment

I am deeply grateful to my supervisors Drs.Giuliano Antoniol and Yann-Gaël Guéhéneuc for their guidance and support.

References

[1] F. Deißenböck 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] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. on Software Engineering*, vol. 28, pp. 970–983, Oct 2002.

[7] J. I. Maletic, G. Antoniol, J. Cleland-Huang, and J. H. Hayes, "3rd international workshop on traceability in emerging forms of software engineering (tefse 2005)." in *ASE*, 2005, p. 462.

[8] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing." in *Proceedings of the International Conference on Software Engineering*, 2003, pp. 125–137.

[9] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[10] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of 22nd IEEE International Conference on Software Maintenance*. Philadelphia Pennsylvania USA: IEEE CS Press, 2006, pp. 469 – 478.

[11] B. Caprile and P. Tonella, "Restructuring program identifier names," in *Proc. of the International Conference on Software Maintenance (ICSM)*, 2000, pp. 97–107.

[12] E. Merlo, I. McAdam, and R. D. Mori, "Feed-forward and recurrent neural networks for source code informal information analysis," *Journal of Software Maintenance*, vol. 15, no. 4, pp. 205–244, 2003.

[13] 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.

[14] 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.

[15] 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.

[16] L. Guerrouj, M. D. 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.

[17] L. Guerrouj, P.Galinier, Y. G. Guéhéneuc, G. Antoniol, and M. Di Penta, "Tris: A fast and accurate identifiers splitting and expansion algorithm," in *Proc. of the Working Conference on Reverse Engineering (WCRE)*, Kingston, 2012, pp. 103–112.

[18] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in *Proc. of the International Conference on Program Comprehension (ICPC)*, Kingston, 2011, pp. 11–20.

[19] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive strategies and looping constructs: an empirical study," *Commun. ACM*, vol. 26, no. 11, pp. 853–860, 1983.

[20] A. von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[21] N. Anquetil and T. Lethbridge, "Assessing the relevance of identifier names in a legacy software system," in *Proceedings of CASCON*, December 1998, pp. 213–222.

[22] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*. IEEE Computer Society, 2009, pp. 158–167.

[23] M.-A. D. Storey, *A Cognitive Framework For Describing And Evaluating Software Exploration Tools*. PhD thesis Simon Fraser University, 1998.

[24] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study." *IEEE Trans. Software Eng.*, vol. 30, no. 12, pp. 889–903, 2004.

[25] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. Portland, Oregon, USA: ACM Press, 2006, pp. 1–11.

[26] J. Sillito, G. C. Murphy, and K. D. Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, pp. 434–451, 2008.