A Case Study of Using Domain Analysis for the Conflation Algorithms Domain

Okan Yilmaz, *Student Member*, IEEE, William Frakes, *Member,* IEEE

***Abstract*—This paper documents the domain engineering process for much of the conflation algorithms domain. Empirical data on the process and products of domain engineering were collected. Six conflation algorithms of four different types: three affix removal, one successor variety, one table lookup, and one n-gram were analyzed. Products of the analysis include a generic architecture, reusable components, a little language and an application generator that extends the scope of the domain analysis beyond previous generators. The application generator produces source code for not only affix removal type but also successor variety, table lookup, and n-gram stemmers. The performance of the stemmers generated automatically was compared with the stemmers developed manually in terms of stem similarity, source and executable sizes, and development and execution times. All five stemmers generated by the application generator produced more than 99.9% identical stems with the manually developed stemmers. Some of the generated stemmers were as efficient as their manual equivalents and some were not.**

***Index Terms*-- Software reuse, domain analysis, conflation algorithms, stemmers, application generator.**

I. Introduction

In the early 1980s software companies started the systematic reuse process through domain engineering to improve software productivity and quality. There has been insufficient empirical study of the domain engineering process and domain products such as reusable components and generators. This paper addresses this problem by documenting and empirically evaluating a domain engineering project for the conflation algorithms domain. This domain is important for many types of systems such as information retrieval systems, search engines, and word processors. The application generator developed for this study extends the domain scope compared to previous ones.

A. Conflation Algorithms Domain

Conflation algorithms are used in Information Retrieval (IR) systems for matching the morphological variants of terms for efficient indexing and faster retrieval operations. The conflation process can be done either manually or automatically. The automatic conflation operation is also called stemming. Frakes [1] categorizes stemming methods into four groups:

Manuscript received July 9, 2007.

affix removal, *successor variety*, *n-gram* and *table lookup*. Affix removal is the most intuitive and commonly used of these algorithm types. In order to determine the stem, affix removal algorithms remove suffixes and sometimes also prefixes of terms. Successor variety and n-gram methods analyze a word corpus to determine the stems of terms. Successor variety bases its analysis on the frequency of letter sequences in terms, while n-gram conflates terms into groups based on the ratio of common letter sequences, called n-grams. Table lookup based methods use tables which map terms to their stems.

We did a domain analysis for the semantic automatic conflation algorithms domain. We analyzed 3 affix removal stemmers, a successor variety stemmer, an n-gram stemmer, and a table lookup stemmer. Based on this analysis, we created a generic architecture, determined reusable components, and designed and developed a little language and an application generator for this domain. We compared the performance of the automatically generated algorithms with their original versions and found that automatically generated versions of the algorithms are nearly as precise as the original versions.

This paper is organized as follows. We present the research in the next section. We describe the domain analysis method we used in this work in Section III. In Section IV, Section V, and Section VI, we present our domain analysis process, products, and process logs. In the next section we analyze the time spent in each step of domain analysis. Section VII shows the evaluation results of the automatically generated stemmers and Section VIII summarizes results and proposes some future research directions.

II. Related Work

Recently several domain engineering approaches and domain analysis methods have been

O. Yilmaz and W. Frakes are with the Computer Science Department, Virginia Tech, 7054 Haycock Road, Falls Church, VA 22043, USA, (e-mail: {oyilmaz, frakes}@vt.edu).

proposed [2]-[9] and surveyed [10]. In this study, we use the *DARE: Domain analysis and reuse environment* [4]. Using DARE, domain related information is collected in a domain book for the conflation algorithms domain.

In information retrieval systems there is a need for finding related words to improve retrieval effectiveness. This is usually done by grouping words based on their stems. Stems are found by removing derivational and inflectional suffixes via stemming algorithms. The first affix removal stemming algorithm was developed by Lovins [12]. This algorithm did stemming by iteratively removing longest suffixes satisfying predefined suffix rules. Several other longest match affix removal algorithms have been developed since [13]-[17]. Porter algorithm [15],[16] is most commonly used because of its simplicity of implementation and compactness. Later Paice [17] proposed another compact algorithm. Hafer and Weiss [18] took a different approach in their successor variety stemming algorithm and proposed a word segmentation algorithm which used successor and predecessor varieties to determine fragmentation points for suffixes. Successor and predecessor varieties are the numbers of unique letters after and before a substring in a corpus. Their algorithm applied several rules to identify the stem from the substrings of each word that appeared in a corpus. The successor variety algorithm has the advantage of not requiring affix removal rules that are based the on the morphological structure of a language. However, the effectiveness of this algorithm depends on the corpus and on threshold values used in word segmentation. Adamson and Boreham [19] developed the N-gram algorithm that uses the number of distinct and common n-character substrings to determine if two or more corpus words can be conflated into the same group. Similar to successor variety, the strength of this algorithm depends on the corpus and the cutoff similarity value chosen. More recently, Krovetz [20] developed the K-stem algorithm that does a dictionary lookup after applying affix removal rules removing inflectional suffixes.

Conflation algorithms have been analyzed and evaluated in several studies [21]-[24]. Lennon et al. [21] compared a number of conflation algorithms used in text-based information retrieval systems in terms of compression percentage and retrieval effectiveness. They found relatively small differences among the conflation methods in terms of these measures. Hull [22] examined the strength of average precision and recall metrics in evaluation of stemming algorithms and proposed several novel approaches for evaluation of stemming algorithms. He did a detailed statistical analysis to better understand the characteristics of stemming algorithms. Frakes et al. [23] analyzed four stemming algorithms in terms of their strength and similarities. They used the Hamming distance measure as well as other commonly used measures. Fox et al. [24] reported an application generator using finite state machines for longest match stemming algorithms. They generated computationally efficient stemmers for Porter [15], Paice [17], Lovins [12] and S-removal [25] stemming algorithms and compared their performance with the developed versions of these algorithms. This paper extends the scope of analysis to other sub-domains of conflation algorithms by analyzing not only affix removal but also successor variety, n-gram, and dictionary lookup types of algorithms.

For this paper we analyzed Lovins, Porter, and Paice as examples of longest match affix removal, and Successor Variety [18], N-gram [19], and K-stem [20] as instances of the remaining three types of conflation algorithms. As the result of the domain analysis we developed an application generator and generated stemmers for Porter, Paice, Lovins, successor variety, S-removal[25], and K-stem algorithms and compared generated algorithms with the corresponding algorithms developed by humans.

III. DARE Domain Analysis Method

In this study we used the DARE domain analysis method and organized the domain information of conflation algorithms in a DARE domain book. This book holds all the domain information that was analyzed and generated. The major sections of the domain book were as follows:

- *Source information subsection* included documents related to the conflation algorithms domain: source code, system descriptions, system architectures, system feature tables, and source notes of the six conflation algorithms that we analyzed
- *Domain scope subsection* contained inputs, outputs, functional diagrams of conflation algorithms that were analyzed as well as a generic functional diagram that we developed as a result of domain analysis
- *Vocabulary analysis subsection* had basic vocabulary information, a facet table for the domain, a synonym table, a domain template, domain thesaurus document, and vocabulary notes
- *Code analysis subsection* showed source code analysis results for conflation algorithms that were analyzed
- *Architecture analysis subsection* contained a generic architecture diagram
- *Reusable components subsection* contained the components that were determined as reusable as the result of domain analysis process
- *Little language subsection* proposed a domain specific language created in Backus-Naur form
- *Application generator subsection* included application generator notes and the source code produced as a product of the conflation algorithms domain analysis

IV. DOMAIN ANALYSIS PROCESS

We started the domain analysis process by gathering source information. We collected published papers in the conflation algorithms subject area as domain documents and the source code of conflation algorithms for system architecture analysis. After building expertise about the conflation algorithms domain we filled out system description questionnaires for each one of these algorithms. Important components of our domain analysis process are in the following subsections.

TABLE IV-1 SYSTEM FEATURE TABLES FOR THE CONFLATION ALGORITHMS

Algorithm Name	Corpus Usage	Dictionary Usage	Natural Language	Type	Stem Generation	Strength
Porter	No	No	English	Longest Match Affix Removal	Yes	Medium
Paice	No	No	English	Longest Match Affix Removal	Yes	High
Lovins	No	No	English	Longest Match Affix Removal	Yes	Medium
Successor Variety	Yes	No	Any	Successor Variety	Yes	Low
N-Gram	Yes	No	Any	N-Gram	No	N/A
K-Stem	No	Yes	English	Dictionary based Inflectional	Yes	Medium

A. Feature Table

We summarized key features of the conflation algorithms in a generic feature table as shown in Table IV-1. Among these algorithms only successor variety and N-gram used a corpus. K-stem was the only algorithm that used a dictionary, and the N-gram was the only algorithm that did not generate stems. Instead of generating stems, the N-gram conflated words into word clusters. Since the successor variety and the N-gram algorithms did not use the morphological properties of the English language, they could be used for conflating words in other natural languages as well.

A stronger stemmer tends to conflate more words into a single class than a weaker stemmer.

Paice has been found to be the strongest algorithm, whereas successor variety was the weakest [23].

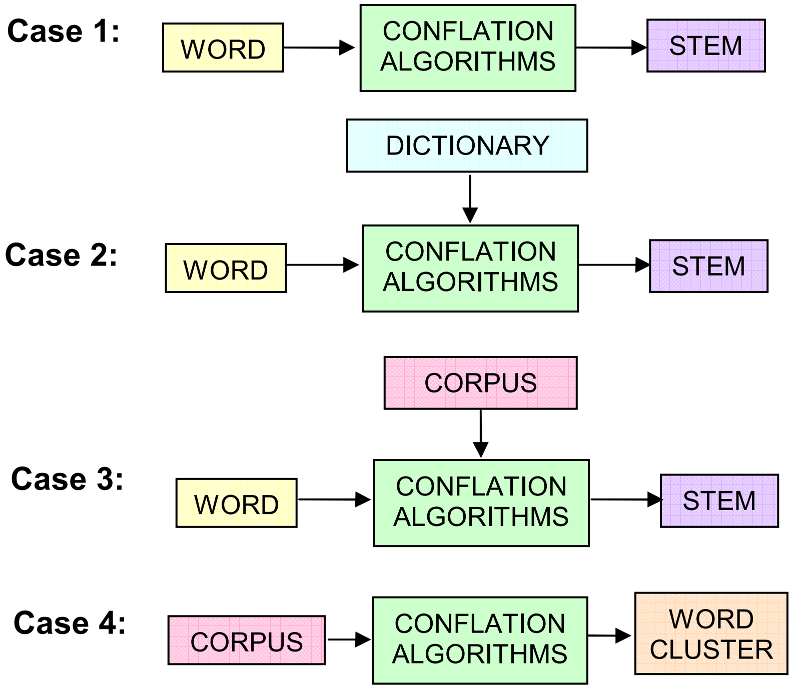

Figure IV-1: Functional diagrams for conflation algorithms

B. Functional Diagrams

To determine the scope of our analysis we categorized the conflation algorithms into four categories and developed a generic functional diagram which combined the functional diagrams of all algorithms analyzed into a generic diagram. Figure IV-1 shows four types of functional diagrams that the analyzed conflation algorithms have.

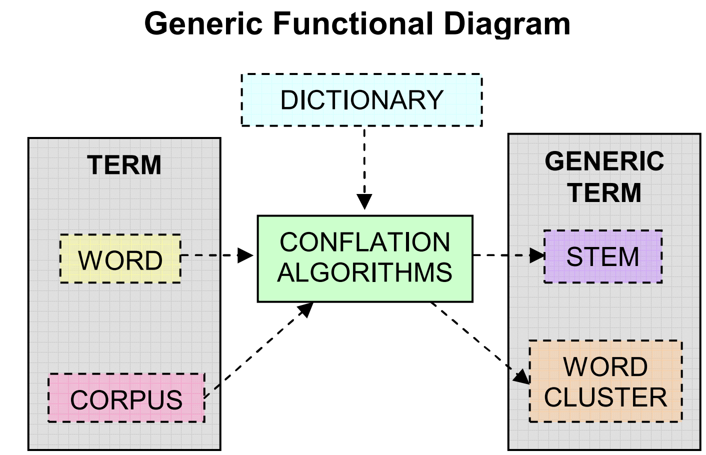

Figure IV-2: Generic functional diagram for conflation

On the other hand, Figure IV-3 presents a generic functional diagram which represents all the algorithms that were analyzed. In this generic architecture diagram dashed lines indicate optional components. As shown in this figure none of these inputs or outputs was common in all algorithms.

C. Scope

Domain scoping was a crucial step in the domain analysis process [11]. In scoping, we determined the functional representation of the domain and bounded the extent of the domain analysis and domain implementation process. The following was the scope of our domain analysis:

- GT←CA(T)

where

- T represents a set of terms.
- GT represents a set of related terms.
- CA is the function that maps T to GT.

In other words, conflation algorithms do a mapping from a set of terms to a set of clusters of generic terms representing common concepts.

The scope of the domain implementation was set to conflation algorithms that determine the stems of words; namely affix removal, successor variety, and table lookup type conflation algorithms.

TABLE IV-2 FACET TABLE FOR CONFLATION ALGORITHMS

Operations	Word parts	Morpheme relations	Document types	Rule Type	Performance
Match	Word	Successor	Abstract	Inflectional	Storage-overhead
Longest Match	Morpheme	Consecutive	Document	Derivational	Under-stemming
Shortest Match	Phoneme	Similarity	Corpus	Affix removal	Over-stemming
Partial Match	Letter	Unique	Information	Successor variety	Weak stemming
Conflation/Conflate	Prefix	Equivalent	Title	Common n-gram	Strong stemming
Removal/Remove	Suffix	Entropy	Description	Cutoff Method	Effectiveness
Recode	Infix	Double	Identifier	Peak and Plateau Method	Correctness
Compare	Affix	Substring	Vocabulary	Complete word method	Recall
Append	Substring	Dice coefficient	Dictionary	Entropy method	Precision
Retrieve	Segment		Manual		Performance
Search	Stem		Stoplist		Outperform
Lookup	Di-gram		Exception list		Experimental evaluation
Right truncation	N-gram				Evaluation
Internal truncation	Character				Measurement
Cut	Root				Equivalent
Dictionary	Term				Sign-test
Compression	Utterance				T-test
Cutoff	Vowel				
Clustering	Consonant				
Measure	Vowel sequence				
Intact	Consonant sequence				
Double	A, e, i, o, u, y				
Length					

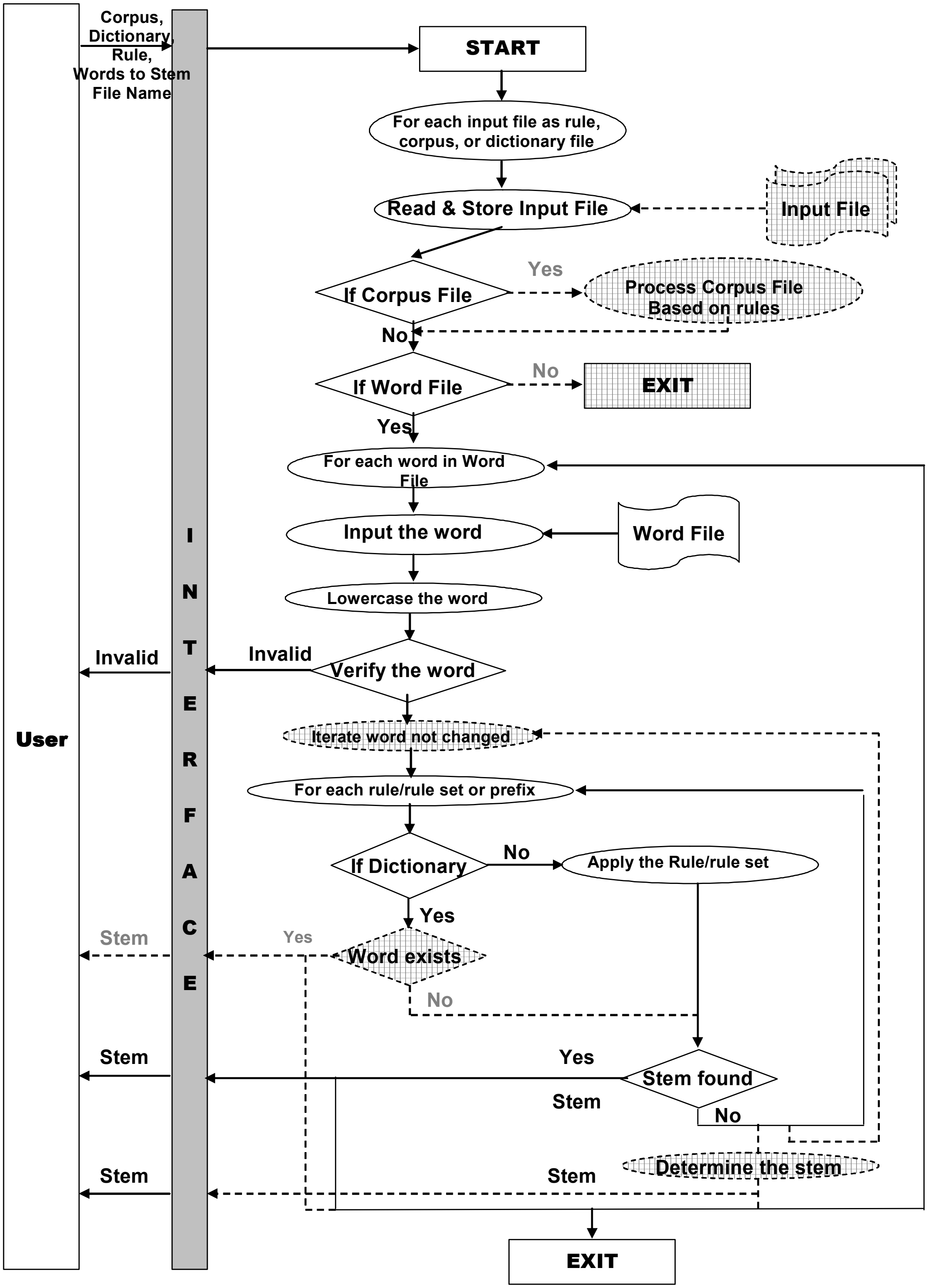

Figure IV-3 Generic System Architecture

D. Vocabulary Analysis

We did a manual vocabulary analysis as well as an automatic analysis by using a web based text analysis tool [27]. We created a facet table from the basic vocabulary generated by vocabulary analysis. Table IV-2 presents the facet table grouping the basic vocabulary into six categories. The facet categories were later used in the little language development process as operations, operands, etc.

E. System Architectures

C and C++ source code of algorithms was analyzed manually and by using *cflow* and the *source navigator* programs on the HP-Unix environment. The system architectures were created on the basis of source code analysis. After analyzing the system architectures of each conflation algorithm we created the generic system architecture. Figure IV-3 shows the generic system architecture. In the generic system architecture diagram optional components are shown with dashes.

V. Domain Analysis Products

After creating the facet table and the generic system architecture, we identified the reusable components, created a little language, and developed an application generator. These three products of our domain analysis are presented in the following subsections.

A. Reusable Components

The reusable components are summarized in Table V-1. These components can be classified into two groups: file processing operations and word part processing operations. All these operations are used as operations in the little language.

B. A Little Language

Using the generic system architecture and reusable components, we represented the operations in Backus–Naur Form. See appendix A for the little language.

TABLE V-1 REUSABLE COMPONENTS OF CONFLATION ALGORITHMS DOMAIN

Reusable Component Category	Operations
Hash Table operations	initialize, search and retrieve, add, delete
Text file operations	open, close, read line
String manipulation operations	substring, string comparison, lowercase, uppercase, string length
String/character validation operations	is AlphaNumeric, is Vowel, is Consonant, shorter than, longer than
File processing/storage operation	Read & store each word from in an input file (e.g. corpus, dictionary)
Word verification operations	check the size, check if it is alphanumeric, etc.
Suffix removal rules	remove a suffix if it is equal to a morpheme
Suffix recode rules	replace a suffix if it is equal to a morpheme

C. Application Generator

After creating the little language for conflation algorithms, we developed an application generator that implemented the operations formally defined in the little language. In order to test the application generator, we generated the source code for Porter, Paice, Lovins, Successor Variety, S-removal and K-stem algorithms by using the application generator.

The application generator was developed to generate code in object oriented fashion from a rule file, where each file consisted of three parts: rules, steps, and main routine. Each rule started with a rule name and specified an operation after the name. For example, the following line from Porter algorithm specified a rule named `1a-r1` which replaced suffix “-*sses*” with “-*ss*”:

```
1a-r1 replace suffix sses ss
```

After the rules part, steps of the algorithm were defined. Each step defined the order of rules to be applied as well as the operations to be performed as a result. For example, the following step from Porter algorithm applied rules `1a-r1, 1a-r2, 1a-r3, 1a-r4` and in case of success it updated the stem of the word analyzed.

```
BEGIN step 1a
  if 1a-r1
  then set word stem
  else if 1a-r2
  then set word stem
  else if 1a-r3
  then set word stem
  else if 1a-r4
  then set word stem
END step 1a
```

The last section in a rule file was the main routine which specified the steps that would be executed:

```
BEGIN stem
  call step 1a
  call step 1b
  call step 1c
  call step 2
  call step 3
  call step 4
  call step 5a
  call step 5b
END stem
```

VI. Domain Analysis Logs

During the domain analysis process we recorded the time for each step in a log file. Table VI-1 summarizes the times spent for each step of domain analysis process. In the upper part of the table each step is presented in detail while in the lower part these steps are classified in higher level categories. Most of the time was spent developing the application generator. This took approximately 80 hours. We spent approximately half of our domain analysis time on the application generator development step. The second and third most time consuming processes were the source collection and analysis, and generic architecture development steps that took about 17 and 15 hours, respectively. Software analysis and development were the most time consuming steps. These results are consistent with the main reason for domain engineering, since an application generator is an investment to save time in the future conflation algorithm development.

The times for the various domain engineering activities were as follows.

TABLE VI-1 TIME SPENT IN EACH STEP OF DOMAIN ANALYSIS PROCESS.

Step Name	Time Spent (hours)
Source collection	13
Learning the concepts	3
Manual code analysis	5
Facet table	2
Expert forms	0.5
Domain book maintenance	2
System architectures	8
System feature tables	0.5
Generic architecture	4
Automatic code analysis	1
Source notes	3
Vocabulary notes	1
Domain scoping	3
Generic feature table	1
Architecture notes	3
Application generator notes	0.5
Glossary	5
Little language	8
Application generator development	80
Review and corrections	4
Category Name	**Time Spent (hours)**
Vocabulary analysis	11
Source collection and analysis	17
Architecture analysis and generic architecture development	15
Domain scoping	4.5
Little language development	8
Application generator development	80.5
Other (domain book generation, etc)	11.5

VII. EVALUATION OF GENERATED STEMMERS

In this section, we evaluated the application generator we developed by comparing the stemmers generated by the application generator with the stemmers developed by humans in terms of the following characteristics of stemmers:

- Similarity of stems produced
- Time spent during development
- Size of the executable of the stemmer
- Number of lines of code (LOC)
- Total execution time

We also compared the box plots of elapsed stemming times of each word in the test set for each version of analyzed stemmers.

A. Evaluation Method

To evaluate the performance of stemmers we needed a test data set. We created a corpus containing 1.15 million words by combining about 500 articles from Harper's Magazine [28], Washington Post Newspaper [29], and The New Yorker [30] with a sample corpus of spoken, professional American-English [31]. We generated a test file with about 45000 unique entries of this corpus by using the text analysis functionality of the application generator. We evaluated, developed, and generated versions of Porter, Paice, Lovins, S-removal, and K-stem stemmers. All these algorithms were in the Perl programming language except for the developed version of the K-stem algorithm which was in C. While the code generated by the application generator was object oriented, the developed versions of these algorithms were not. During the evaluation process we verified the stems generated by these stemmers and fixed bugs in the developed code and in rule files for the application generator.

B. Evaluation Criteria

We tested the performance of our application generator by comparing the generated stemmers to the stemmers developed by humans in terms of stem similarity, development time, executable size, number of LOC, and total execution time.

C. Evaluation Results

TABLE VII-1 COMPARISON RESULTS FOR DEVELOPED AND GENERATED STEMMERS

Algorithm Name	Porter		Lovins		Paice		S-removal		K-stem	
Stem Similarity	Identical	Different	Identical	Different	Identical	Different	Identical	Different	Identical	Different
	45006	1	45006	1	45007	0	45007	0	44970	37
Development Time (hours)	Generated	Developed	Generated	Developed	Generated	Developed	Generated	Developed	Generated	Developed
	4	12	3	NA	6	NA	0.5	0.5	6	NA
Executable Size (bytes)	Generated	Developed	Generated	Developed	Generated	Developed	Generated	Developed	Generated	Developed
	849247	390778	900851	398528	874039	393640	839315	387443	856334	334689
Number of LOC	Generated	Developed	Generated	Developed	Generated	Developed	Generated	Developed	Generated	Developed
	453	126	1180	555	1069	1039	142	36	719	2035
Execution Time (seconds)	Generated	Developed	Generated	Developed	Generated	Developed	Generated	Developed	Generated	Developed
	3.03	1.52	6.58	1.73	2.72	6.66	0.70	0.44	3.41	1.02

We used a test file of 45007 unique entries in our experiments. Table VII-1 summarizes the evaluation results. All five stemmers generated by the application generator produced more than 99.9% identical stems with the developed stemmers. Preparing the rule file for Porter algorithm took 4 hours while developing the same algorithms took 12 hours. Since the S-removal is a very simple stemming algorithm, both developing it and generating rule files for it took about half an hour. For the rest of the algorithms we report the rule file generation time since we did not have information about their actual development time. Executables generated from the Perl scripts of all generated stemmers were at least twice as big as the developed stemmers. Among all algorithms developed K-stem had the smallest executable. This was partly because it was developed in C rather than Perl. On the other hand, for the same reason developed K-stem had highest LOC among all stemmers. The generated stemmers were slower than the developed ones except for the Paice algorithm. We did not find a statistically significant difference between the generated and developed stemmers in terms of LOC and execution time due to the limited number of algorithms tested.

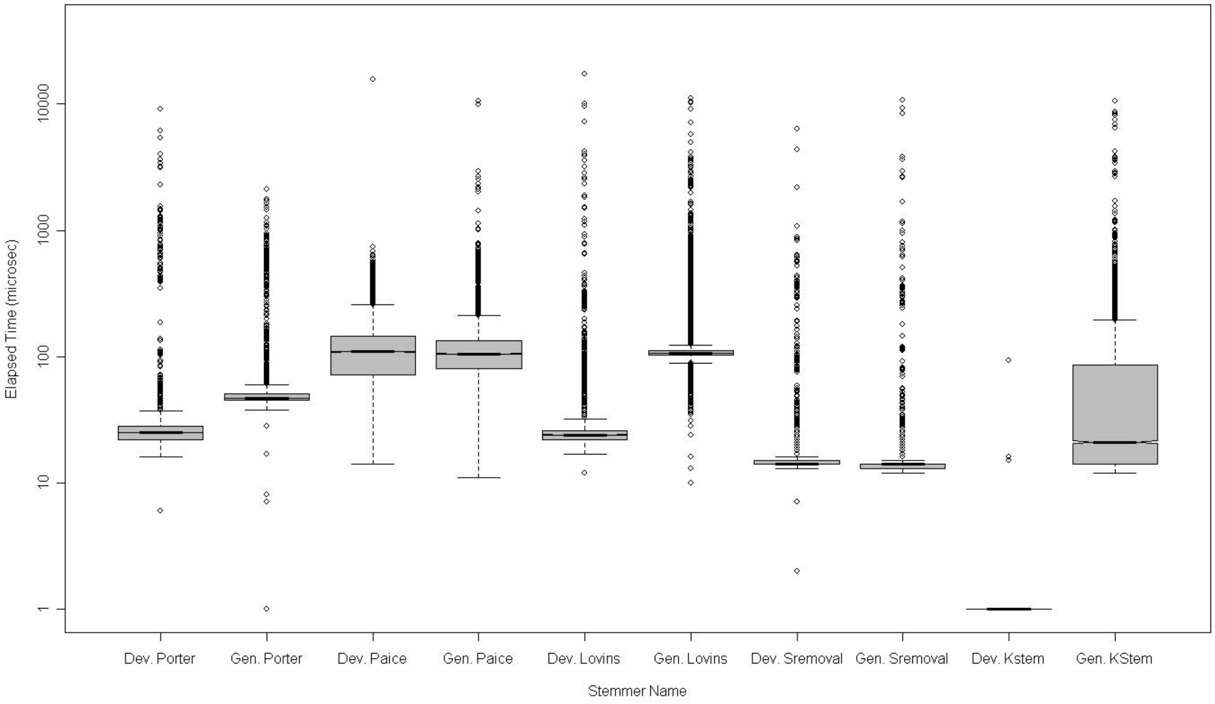

Figure VII-1 Box Plot Diagrams for Elapsed Stemming Times of Stemmers in Log. Scale.

D. Analysis of Elapsed Stemming Times of Generated and Developed Stemmers

Stemming times per word stemmed is reported in the box plot for each stemmer in Figure VII-1. Developed K-Stem and Developed Paice had the lowest and highest average elapsed stemming times respectively. Generated stemmers for Paice and S-removal performed a little better than developed ones. On the other hand, developed Porter, Lovins, and K-stem performed much better than the generated versions of these algorithms. Although the total time spent by developed K-Stem was more than the developed and generated versions of S-removal stemmers, the average elapsed time for each word stemmed turned out to be much shorter. This was because the time spent during the dictionary reading was not included in the elapsed time for stemming each word. Figure VII-1 shows many outliers for each stemmer. We stemmed the data set several times and compared the outliers in each case to determine the characteristics of these outliers. We saw that in each run we had different outliers and concluded that the outliers were not caused by the stemming algorithms or stemmers. Stemming operations were normally very fast operations taking less than 100 microseconds on the average. When the test was running, the other operations done by the windows operating system were affecting our result by causing several outliers.

VIII. Conclusions and Future Work

In this paper we presented a case study of using domain analysis for the semantic conflation algorithms domain. We analyzed Porter, Lovins, Paice, Successor variety, N-gram, and K-stem algorithms and as products of our domain analysis we determined the reusable components, created a little language in Backus–Naur Form, and developed an application generator in the Perl language. Using the application generator, we generated Perl code for Porter, Paice, Lovins, S-removal, and K-stem algorithms. We compared the performance of stemmers generated by the application generator with the corresponding stemmers developed by humans in terms of identical stem generation, development times, size of executables, number of LOC, and the total time spent to stem all terms in our test set. We created and used a corpus with about 45000 words to test stemmers. Our results indicated that the stems produced by the generated and developed stemmers produced identical results for more than 99.9% of evaluations. We also determined that stemmers produced by application generators have bigger executables than the stemmers developed by humans. We did not find a statistically significant difference between the generated and developed stemmers in terms of LOC and the total time spent to stem all terms in the test set due to the limited number of algorithms tested. We also analyzed elapsed stemming times of these developed and generated stemmers. We presented a box plot diagram for each stemmer in terms of the elapsed stemming times. We determined that generated stemmers performed better for some cases and worse in some other cases.

In this study we have done a domain engineering project for affix removal, successor variety, n-gram, and table lookup types of stemming algorithms and generated code for all types other than the N-gram algorithm. In future work, we plan to generate a stemmer for N-gram as well. Also we did not compare the successor variety stemmer with a successor variety stemmer developed by humans, but hope to do this in the future.

Appendix

A. The Little Language

TABLE VIII-1 THE LITTLE LANGUAGE DEVELOPED FOR CONFLATION ALGORITHMS DOMAIN

Little Language Rules

```
letter = "a".."z";
digit = "0".."9";
number = digit {digit};
char = letter | digit | “-“;
morpheme = letter {letter};
morph_name = "word" | "stem";
succ_name = "pred" | "succ" | "this";
place = "suffix" | "prefix";
any_place = place | "any";
some_place = any_place | "all";
margin_place = "first" | "last";
any_margin_place = margin_place | "any";
comparison = "=" | ">" | ">=";
all_comparison = comparison | "<" | "<=" | "==";
alpha_char = "consonant" | "consonantny" | "vowel" | "vowelny" | letter;
double_char = ["double"] alpha_char | "or" double_char;
name = char {char};

rule_name = name;
step_name = name;
```

```
rule = rule_name "append" place morpheme |
      rule_name "replace" some_place morpheme morpheme |
      rule_name "remove" (some_place morpheme | margin_place) |
      rule_name "isVowel" morph_name any_margin_place |
      rule_name "isConsonant" morph_name any_margin_place |
      rule_name "isMeasure" morph_name comparison number |
      rule_name ("isa" | "match") any_place double_char |
      rule_name "length" morpheme all_comparison number;

segment_opr = "add" name | "get" | "reset";

succ_var_rules = "cutoff" |
           "successor" succ_name all_comparison succ_name |
        "entropy" succ_name all_comparison number;

sys_rule = "lookup" morph_name | "intact" | succ_var_rules;

CR = "\n";

rules = rule CR {rule CR};

if_rules = rule_name |
      sys_rule |
      if_rules "or" if_rules |
      if_rules "and" if_rules |
      "not" if_rules;

sys_oper = "set" morph_name morph_name |
      "set" "stop" number |
      "set" name "length" morph_name |
      "call step" step_name |
      "break" |
      "segment" segment_opr |
      sys_oper "and" sys_oper;

then_oper = sys_oper | "if" if_rules CR "then" sys_oper;

else_oper = sys_oper | if_oper;

if_oper = "if" if_rules CR "then" then_oper CR [ "else" else_oper CR ];

step = if_oper | sys_oper CR | if_oper step;

iterate = "BEGIN iterate" CR step CR "END" "iterate";
for_each = "BEGIN for-each prefix" CR step CR "END" "for-each prefix";
stem_body = iterate | for_each | step;
all_rules = "BEGIN rules" CR rules CR "END" "rules" CR;
each_step = "BEGIN step" step_name CR step "END step" step_name CR;
all_steps = each_step {each_step};
stem_steps = "BEGIN stem" CR stem_body "END stem" CR;

# rule file is a sequence of all_rules all_steps and stem_steps
rule_file = all_rules all_steps stem_steps;
```

REFERENCES

[1] Frakes, W. B. “Stemming algorithms.” In: Frakes, W.B. and R. Baeza-Yates, (editors): *Information Retrieval: Data Structures and Algorithms,* Englewood Cliffs, NJ: Prentice-Hall, 1992.

[2] Prieto-Diaz, R., “Reuse Library Process Model. Final Report,” Technical Report Start Reuse Library Program, Contract F19628–88-D-0032, Task IS40, Electronic Systems Division, Air Force Command, USAF, Hanscomb AFB, MA, 1991.

[3] Simos, M., R. Creps, C. Klingler, and L. Lavine, “Organization Domain Modeling (ODM) Guidebook, Version 1.0,” Unisys STARS Technical Report No. STARS-VC-A023/011/00, STARS Technology Center, Arlington, VA, 1995.

[4] Frakes, W., Prieto-Diaz, R. & Fox, C. J. “DARE: Domain analysis and reuse environment.” *Annals of Software Engineering*, Vol. 5, 125-141, 1998.

[5] D.M. Weiss and C.T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[6] K.C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering," *IEEE Software*, vol. 19, no. 4, pp. 58-65, July/Aug. 2002.

[7] C. Atkinson et al., *Component-Based Product Line Engineering with UML*. Addison-Wesley, 2002.

[8] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.

[9] R. Ommering et al., “The Koala Component Model for Consumer Electronics Software,” *Computer*, vol. 33, no. 3, pp. 78-85, Mar. 2000.

[10] Frakes, W. and Kang, Kyo, Software Reuse Research: Status and Future, *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp 529-536, July 2005.

[11] Frakes, W. "A Method for Bounding Domains". *Proceedings of the IASTED International Conference Software Engineering and Applications,* Las Vegas, NV, Nov, 2000.

[12] Lovins, J. B. "Development of a stemming algorithm." *Mechanical Translation and Computational Linguistics*, Vol. 11, 22-31, 1968.

[13] G. Salton, “Automatic information organization and retrieval”, Mc Graw Hill, New York, 1968.

[14] Dawson, J. L. “Suffix removal and word conflation,” *ALLC Bulletin*, Vol. 2(3), 33-46, 1974.

[15] Porter, M. “An algorithm for suffix stripping.” *Program*, Vol. 14, Number 3, 130-137, 1980.

[16] http://www.tartarus.org/~martin/PorterStemmer/

[17] Paice, Chris D. "Another stemmer." *SIGIR Forum*, Vol. 24 (3), 56-61, 1990.

[18] Hafer, M., and Weiss, S. “Word segmentation by letter successor varieties,” *Information Storage and Retrieval*, Vol. 10, 371-85, 1974.

[19] Adamson, G. and Boreham J. “The use of an association measure based on character structure to identify semantically related pairs of words and document titles, “ *Information Storage and Retrieval*, Vol. 10, 253-60, 1974.

[20] Krovetz, R., 1993: “Viewing morphology as an inference process”, in R. Korfhage, E. Rasmussen & P. Willett (eds.), *Proceedings of the 16th ACM SIGIR conference*, Pittsburgh, PA, pp.191-202, June-July, 1993.

[21] Lennon, J. B., Pierce, D. S., Tarry, B. D., and Willet, P. “An evaluation of some conflation algorithms for information retrieval”, *Journal of information Science*, Vol. 3, 177-183, 1981.

[22] Hull, D. A. “Stemming algorithms: A case study for detailed evaluation.” *Journal of the American Society for Information Science*, Vol. 47(1), 70-84, 1996.

[23] Frakes, W.B. and Fox, C. J. “Strength and similarity of affix removal stemming algorithms” *SIGIR Forum*, Vol. 37, 26-30, Spring 2003.

[24] Fox, B. and Fox, C. J. “Efficient Stemmer generation” *Information Processing and Management: an International Journal,* Vol. 38 , Issue 4, 547–558, 2002.

[25] Harman, D. “How Effective is Suffixing?” *Journal of the American Society for Information Science,* 42(1), 7-15, 1991.

[26] Kang, K., Cohen S., Hess J., Novak, W. and Peterson S., “Feature-Oriented Domain Analysis (FODA) feasibility study,” Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, 1990.

[27] http://textalyser.net

[28] http://www.harpers.org

[29] http://www.washingtonpost.com

[30] http://www.newyorker.com

[31] http://www.athel.com/sample.html

[32] Downloaded from http://www.cmcrossroads.com/bradapp/clearperl/sclc-cdiff.html

Okan Yilmaz (M’05) received his Bachelor of Science and Master of Science degrees in computer engineering and information science from Bilkent University, Ankara, Turkey. He is currently a Ph.D. student at Virginia Tech. His research interests include software engineering, wireless communications, multimedia, and mobile computing.

William B. Frakes received the BLS degree from the University of Louisville, the MS degree from the University of Illinois at Urbana-Champaign, and the MS and PhD degrees from Syracuse University. He is an associate professor in the computer science department at Virginia Tech. He chairs the IEEE TCSE committee on software reuse, and is an associate editor of IEEE Transactions on Software Engineering.