Concept-Based Failure Clustering

Nicholas DiGiuseppe
University of California, Irvine
Department of Informatics
nicholas.digiuseppe@uci.edu

James A. Jones
University of California, Irvine
Department of Informatics
jajones@ics.uci.edu

ABSTRACT

When attempting to determine the number and set of execution failures that are caused by particular faults, developers must perform an arduous task of investigating and diagnosing each individual failure. Researchers proposed failure-clustering techniques to automatically categorize failures, with the intention of isolating each culpable fault. The current techniques utilize dynamic control flow to characterize each failure to then cluster them. These existing techniques, however, are blind to the intent or purpose of each execution, other than what can be inferred by the control-flow profile. We hypothesize that semantically rich execution information can aid clustering effectiveness by categorizing failures according to which functionality they exhibit in the software. This paper presents a novel clustering method that utilizes latent-semantic-analysis techniques to categorize each failure by the semantic concepts that are expressed in the executed source code. We present an experiment comparing this new technique to traditional control-flow-based clustering. The results of the experiment showed that the semantic-concept clustering was more precise in the number of clusters produced than the traditional approach, without sacrificing cluster accuracy.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

Keywords

Fault Clustering, Debugging, Testing

1. INTRODUCTION

Identifying when software executions fail due to the same fault is a difficult task. Failures are often caused by small defects in a large program [12] and deciphering a "due-to" relationship from program spectra can be laborious [9]. However, identifying "due-to" relationships can save time and resources by parallelizing debugging, preventing redundant work (*i.e.,* two developers unknowingly fixing the same fault) and enabling more accurate automatic localization.

For over two decades, researchers have proposed approaches to perform automatic failure clustering. For example, Podgurski and Dickinson *et al.* used control-flow-based (specifically, branch profiles) techniques (hereafter referred to as *control-based*) [2, 11] to characterize and cluster executions. Liu *et al.* proposed using fault-localization results for the same purpose. Unfortunately, by primarily concentrating on control flow, the semantic intent of the execution (*i.e.,* what is the test case or execution of the software attempting to achieve?) can be limited in its expressiveness. We conjecture that such limitations can limit the ability of the clustering technique to correctly identify shared execution intent among executions that may appear dissimilar based on their control-flow, alone.

This paper presents a *concept-based execution clustering* technique to categorize executions that fail due to the same fault by utilizing latent-semantic analyses (LSA). Such analyses attempt to provide conceptual topics by utilizing natural-language text found in documents. We use such analyses to mine the subset of the source code that was executed by each failure to provide information about its conceptual topics and then cluster based upon those. Our technique approximates the intent of each execution by employing LSA on the natural language used in source code (*e.g.,* variable identifiers and comments) involved in each execution. LSA incorporates a family of techniques that select words from a document that heuristically most represent it. Therefore, we postulate that the words most uniquely appearing in many executed lines of code for a specific execution, represent the execution's intent. While LSA has not been used in failure-clustering, it has found success in both program comprehension (*e.g.,* [8]) and fault localization (*e.g.,* [10]) techniques.

We demonstrate this technique with a preliminary evaluation on real-world software. Our evaluation measures the effectiveness of concept-based clustering in selecting correct clusters and presents a comparison with control-flow-based clustering. The main contributions of this work are:

1. A novel approach to cluster software failures — concept-based clustering. Concept-based clustering leverages natural language to provide clusters of executions with similar semantic intent.
2. A comparative evaluation to assess concept-based clustering and control-flow-based failure clustering. Our

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

evaluation demonstrates increased precision and comparable cluster accuracy for concept clustering.

2. MOTIVATION AND NOVELTY

Traditionally, automated-failure-clustering techniques exclusively employed control-flow profiles (*e.g.,* [11]) or fault localization (*e.g.,* [9]). Podgurski *et al.* found that difficulties arose when distinguishing executions that fail due to the same cause because the executed faults constitute a small portion of a large execution trace [12].

While such profile-based techniques focus entirely upon dynamic control-flow-data, our technique leverages both static and dynamic data to inform our clusters. Our intuition is that executions that fail due to the same cause execute similar program semantics (variables or methods that are topically similar). The novelty of concept-based clustering stems from its approach, choice of input data, and use of data. A concept-based clustering technique attempts to discover an execution's intent by analyzing how natural language is used in executed source code.

This work relies on the lightweight LSA-technique: weighted term frequency with inverse document frequency (TF-IDF). TF-IDF correlates how frequently a word appears in an individual document with how frequently the word appears in the remainder of the corpus. The intuition being, if a word appears often in document X and infrequently in the remaining corpus, then the word uniquely represents X.

We present a motivating example for our approach in Figure 1. This figure exhibits source code on the left, execution coverage on the right, the TF-IDF vectors in the bottom-right (we only show words with at least one non-zero score), and the clusters formed from the TF-IDF clustering and control-flow-based clustering on the bottom-left.

The source code is encoded with various features of "employees," (*e.g.,* seniority, tenure, and citizenship status). This program contains two faults labeled "fault 1" and "fault 2" that occur in the domestic and international employee sections of the code respectively. There are four test cases (T1–T4), all of which are failures, such that T1 and T2 fail due to fault 1, while T3 and T4 fail due to fault 2.

The cosine similarity scores for each test case are shown in the bottom left of the figure. When the profiles of each test case are compared with the cosine similarity metric, no pairwise combinations are more than 50% similar. However, when examining the pairwise TF-IDF comparisons, we find that {T1,T2} and {T3,T4} are 100% similar. Additionally, with TF-IDF, all other pairwise combinations are 0% similar, which is ideal. In this example, depending on the thresholds used to qualify executions as similar, control-flow-based clustering would have not clustered any executions, or incorrectly clustered executions that failed due to different faults. The concept-based clustering would have accurately and precisely clustered the correct executions.

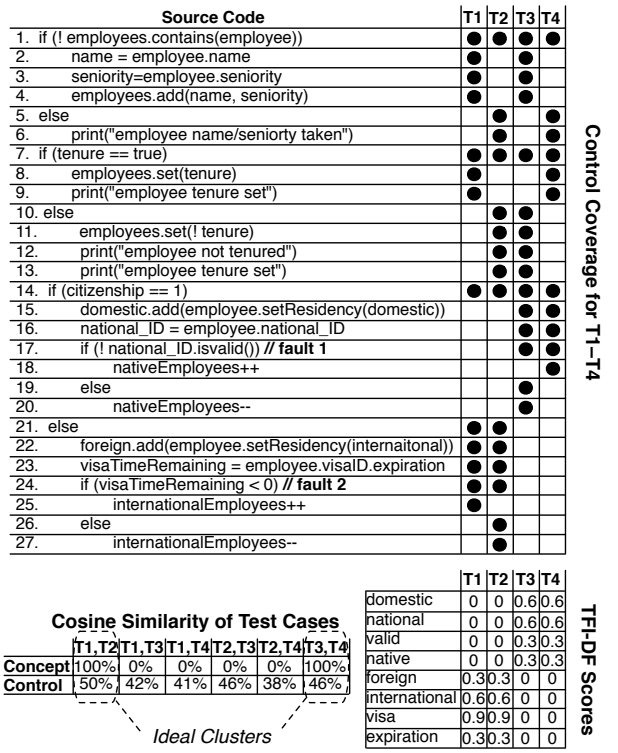

Figure 1: An example of control-flow-based and concept-based clustering. T1 and T2 fail due to fault-2, and T3 and T4 fail due to fault-1.

3. CONCEPT-BASED CLUSTERING

In this section, we describe how to utilize the TF-IDF statistic to represent executions by the concepts (*i.e., words* or *terms*) that are primarily relevant to them. For the TF-IDF statistic, a word's frequency (*i.e., term frequency*, or "TF") is the number of times that a particular term appeared in a particular document. The word's uniqueness is expressed as its *inverse document frequency* (or, "IDF"), which is a measure of how rarely that word appears in the corpus (*i.e.,* the set of all documents). For example, a word that appears in all documents (such as "the") will likely not be important for any documents because its IDF score will be low. These two measures are combined (through multiplication) to produce the TF-IDF statistic to represent how important a given word is to a given document.

Software-engineering researchers have applied this statistic to source code by defining *a source-code file* as the "document" and *all source-code in the program* as the "corpus." In contrast with such static-analysis mapping of concepts, we map these definitions to dynamic concepts. We define the document as the set of lines of code, across all files, that are executed by an individual execution (*i.e.,* a test case). We define the corpus as all executed lines of code, across all files, for all of the executions (*i.e.,* the test suite). We defined the document and corpus in this way to target dynamic execution semantics and to allow the technique to work with programs containing any number of files (few or many).

Concept-based clustering using TF-IDF requires multiple components: a test suite, a statement-level instrumentation tool, the source code, and optionally, textual parsing tools, and a sequence of steps, described as follows.

Step 1: Instrument and execute the program. We instrument the program for statement profiles (*i.e.,* the execution count for each line). Profiles can be gathered with lightweight measures like Gcov[1] or Cobertura.[2]

[1]http://http://gcc.gnu.org

[2]http://cobertura.sourceforge.net

Step 2: Parse the source code. For each line, store the words contained therein. To identify the words in each line of code, textual-based splitters should to be utilized. Our implementation uses a camel-case splitter that separates words based upon their capitalization, though more complex variations exist that exploit domain knowledge (*i.e.,* Samurai [6]). For example, the identifier `isTreeFull` becomes three words: `is`, `tree`, and `full`. Other textual normalization techniques can be applied at this stage (*e.g.,* accounting for variation of verb tense or plurality), but our early prototype implementation currently uses only a splitter.

Step 3: Compute the weighted term frequency (TF). Each execution will have a unique TF score for every word. Term frequency is computed, per execution, for each word by evaluating the sum of all execution instances of any line that contains the word within that execution. For example, consider three executed lines that contain the word "tree." The first is executed once, the second is executed twice, and the third is executed three times. Thus, "tree" has a term frequency of $6 = (1 + 2 + 3)$ for that particular execution. We follow the standard TF-IDF practice to normalize (*i.e., weight*) the term frequency by taking its logarithm to reduce the influence of oft repeated words.

Step 4: Compute the inverse document frequency (IDF). Every execution shares IDF scores for all words. For each word w and the set of all executions E, $IDF(w, E)$ is defined as $\log(\frac{|e \in E : w \in e|}{|E|})$. For example, the word "tree" is contained in instructions that were executed by 25 of the 50 executions — the IDF score for "tree" is thus $\log(25/50)$.

Step 5: Perform hierarchical clustering. Like many traditional clustering methods, an agglomerative or divisive clustering technique is applied. With both techniques one of two stopping mechanisms is in place: (1) stop producing new clusters when a certain preset number of clusters are produced (place all remaining executions into one of those clusters regardless of similarity), or (2) stop producing clusters when no comparison produces a similarity above a certain threshold (regardless of how many clusters are produced).

4. EVALUATION

4.1 Objects, Variables, and Setup

To better understand the quality of clusters produced by concept-based clustering, we performed an experiment with the Unix utility, Sed, version 3.02. Sed contains over 10,000 LOC, has 362 test cases, and contains both seeded and real faults. Sed, its test cases, and faulty versions were taken from the Software-artifact Infrastructure Repository [5]. Twenty versions of Sed were created for this experiment by randomly selecting and including four faults from the set of faults included with the subject.

Our independent variable is the clustering technique used: concept-based (using TF-IDF) or control-flow-based (using statement profiles). The two dependent variables are *cluster purity* and the *quantity of clusters*. Cluster purity measures the degree to which executions within a cluster fail due to the same fault. Cluster purity is measured by evaluating the composition of a cluster. The purity of a cluster C of test cases $t \in C$ is defined using the most represented fault f. Each failing test case was diagnosed to the fault causing its failure through a deductive process that is described in prior work [4]. Cluster purity is defined as $\frac{|t \text{ failing due to } f|}{|t \in C|}$. Ideally, cluster purity would be 100%. However, 100% purity would be simple to achieve if, for N test cases, we produced N clusters, each containing one test case. The *quantity of clusters* metric accounts for this problem. Each version of Sed contains four faults, therefore, an ideal clustering technique would produce exactly four clusters. By analyzing both metrics, we can compare purity (if clusters represent exactly one fault) and precision (if the correct amount of clusters are produced).

Table 1: The results for both clustering techniques.

	Avg. Purity	Avg. Number Clusters
Concept	94%	10.8
Control-flow	95%	40.8

In our experiment, we perform agglomerative clustering and utilize a similarity threshold to determine the point at which clustering should cease. As is standard practice in clustering research, our similarity threshold is determined by training on a random 10% of the executions, and the evaluation is performed utilizing the remaining 90%. The similarity threshold is needed because, in practice, neither developers nor the clustering technique usually cannot know the number of faults that are causing failure, and thus cannot know the ideal number of clusters to produce.

4.2 Results and Analysis

The results of our experiment are exhibited in Table 1. The table presents the average purity and average number of clusters for both techniques. Concept-based and control-flow-based clustering attained similarly pure clusters, 94% and 95% respectively, but differed substantially on the quantity of clusters. Concept-based clustering produced nearly 75% fewer clusters than control-flow-based clustering.

The "extra" clusters in both techniques are redundant clusters representing superfluous instances of the same fault. For example in one faulty, version 14 clusters were produced, seven of which represented the same fault. A team of developers dividing the workload potentially would assume there to be 14 unique faults, resulting in redundant work.

We hypothesize that concept-based clustering produced fewer clusters because it identified similar executions in spite of differing control paths. In circumstances where executions contained similar control paths, the execution semantics are also similar, which results in similar performance for both techniques. However, when control paths differed but execution semantics were similar, only concept-based clustered the test cases correctly. While both techniques are far from ideal (*e.g.,* they produce too many clusters), *in our experiment, concept-based clustering produced fewer clusters without sacrificing purity.*

5. RELATED WORK

This work primarily relates to latent semantic analysis and failure clustering. To the best of the authors' knowledge, no techniques exist that combine these fields, so each research area will be considered separately.

5.1 Latent Semantic Analysis

In the realm of debugging, Andrzejewski *et al.* [1] combines LSA with probabilities to classify "usage topics" and "failure topics." This classification identifies predicates that correlate with failure. Later, Lukins *et al.* [10] matched LSA

generated topics with bug reports to assist in localization. In contrast to these techniques that focus on debugging, our technique uses LSA to automatically cluster failures failing due to the same fault, and makes no attempt to localize or explain them. We envision the above stated techniques to be used after the failures have been clustered.

5.2 Failure Clustering

Early work in failure clustering performed by Podgurski *et al.* evaluated the effectiveness of using execution profiles to correlate failure cause [12] and to isolate features within profiles that identify a failure [11]. Control-flow-based clustering was expanded upon by Dickinson, Leon, and Podgurski who found that "executions which do fail have unusual properties that may be reflected in execution profiles" [3]. The work by Podgurski *et al.* differs from our work in their use of control-flow data as opposed to execution semantics. Our technique attempts to compare a semantic representation of an execution as opposed to control-flow data.

5.3 Author's Previous Work

Jones *et al.* [7] developed two failure clustering techniques and suggested an application of this clustering for enabling a parallelized debugging process among multiple developers. Recently, DiGiuseppe and Jones [4] investigated failure-clustering assumptions to better understand how purity, multi-fault failures, and control flow impact failure clustering. This work leverages a more accurate understanding of failure-clustering assumptions and the concept that metrics beyond control flow can accurately identify failure causes to improve upon traditional clustering techniques.

6. EXPECTED FEEDBACK

We explore leveraging the source code's natural-language as opposed to control flow, because faults are often a very small piece of a much larger execution. We are interested in discussing alternative approaches to representing an execution and in combining existing and future metrics with natural language for a more precise depiction (*i.e.,* combining metrics like statistical fault localization or Markov chains with natural language).

TF-IDF clustering focuses on the executed source to derive its natural language. However, there are many supplementary artifacts and components that could potentially improve our approach. We are interested in discussing the tradeoffs of incorporating auxiliary components (*e.g.,* developer comments) and artifacts (*e.g.,* test-case descriptions, use-case descriptions, and expected-behavior descriptions).

We evaluate TF-IDF clustering using a purity metric and the number of clusters (a type of precision). However, neither of these metrics truly measure "usefulness." Thus, we are interested in discussing different evaluation strategies that take into consideration a cluster's utility for developers in addition to their quality.

7. CONCLUSIONS

In this paper we present a novel approach to cluster software failures — concept-based clustering. Concept-based clustering attempts to represent *execution intent* by evaluating the natural language of executed source-code, and clustering based upon it. We also present a motivating example, which illustrates circumstances where semantics enable better clustering than only control flow metrics. This paper also presents a preliminary evaluation comparing concept-based clustering with traditional control-flow-based clustering. In our experiment, the concept-based approach produced more accurate results by producing fewer clusters without sacrificing purity. Our evaluation serves as a proof of concept that concept-based clustering may have merit for providing accurate and meaningful failure clustering results.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation, through Award CCF-1116943 and through Graduate Research Fellowship under Grant No. DGE-0808392.

9. REFERENCES

[1] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In *European Conference on Machine Learning*, 2007.

[2] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the International Conference on Software Engineering*, 2001.

[3] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *Proceedings of the International Symposium on Foundations of Software Engineering*, 2001.

[4] N. DiGiuseppe and J. A. Jones. Software behavior and failure clustering: An empirical study of fault causality. In *Proceedings of the International Conference on Software Testing*, 2012.

[5] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 2005.

[6] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *International Working Conference on Mining Software Repositories*, 2009.

[7] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *Proceedings of the International Symposium on Software Testing and Analysis*, New York, NY, USA, 2007.

[8] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 2007.

[9] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2006.

[10] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 2010.

[11] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the International Conference on Software Engineering*, May 2003.

[12] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang. Estimation of software reliability by stratified sampling. *ACM Transactions on Software Engineering and Methodologies*, 1999.